ディープラーニングを深く学ぶ

ASTON ZHANG, ZACHARY C. LIPTON, MU LI, AND ALEXANDER J. SMOLA

Contents

1 序文 *page* 9
- 1.1 本書について 9
- 1.2 謝辞 16
- 1.3 要約 17
- 1.4 演習 17

2 インストール 19
- 2.1 Miniconda のインストール 19
- 2.2 ディープラーニングフレームワークと d2l パッケージのインストール 20
- 2.3 コードのダウンロードと実行 20

3 記法 22
- 3.1 数値オブジェクト 22
- 3.2 集合論 22
- 3.3 関数と演算子 23
- 3.4 微積分 24
- 3.5 確率と情報理論 24

4 はじめに 25
- 4.1 動機付けとなる例 26
- 4.2 主要なコンポーネント 28
- 4.3 機械学習の問題の種類 32
- 4.4 ルーツ 46
- 4.5 ディープラーニングへの道 48
- 4.6 成功事例 52
- 4.7 ディープラーニングの本質 54
- 4.8 まとめ 55
- 4.9 練習問題 56

5 前提知識 57
- 5.1 データ操作 57
- 5.2 データ前処理 76
- 5.3 線形代数 80
- 5.4 微分積分 104

5.5 自動微分 111
5.6 確率と統計 122
5.7 ドキュメント 137

6 回帰のための線形ニューラルネットワーク 144
6.1 線形回帰 144
6.2 実装のためのオブジェクト指向設計 158
6.3 合成回帰データ 170
6.4 線形回帰のスクラッチ実装 175
6.5 線形回帰の簡潔な実装 187
6.6 汎化 194
6.7 重み減衰 200

7 分類のための線形ニューラルネットワーク 209
7.1 ソフトマックス回帰 209
7.2 画像分類データセット 218
7.3 分類の基礎モデル 224
7.4 スクラッチからのソフトマックス回帰の実装 228
7.5 Softmax 回帰の簡潔な実装 236
7.6 分類における汎化 241
7.7 環境と分布シフト 248

8 多層パーセプトロン 261
8.1 多層パーセプトロン 261
8.2 多層パーセプトロンの実装 276
8.3 順伝播、逆伝播、および計算グラフ 282
8.4 数値安定性と初期化 287
8.5 ディープラーニングにおける汎化 295
8.6 ドロップアウト 300
8.7 Kaggle で住宅価格を予測する 309

9 ビルダーのためのガイド 319
9.1 層とモジュール 319
9.2 パラメータ管理 332
9.3 パラメータの初期化 339
9.4 遅延初期化 347
9.5 カスタム層 351
9.6 ファイル入出力 358
9.7 GPU 365

10 畳み込みニューラルネットワーク 379
10.1 全結合層から畳み込みへ 380
10.2 画像のための畳み込み 386
10.3 パディングとストライド 398
10.4 複数入力チャネルと複数出力チャネル 405

10.5 プーリング 412
10.6 畳み込みニューラルネットワーク（LeNet） 423

11 現代的な畳み込みニューラルネットワーク 434
11.1 畳み込み深層ニューラルネットワーク（AlexNet） 435
11.2 ブロックを用いたネットワーク（VGG） 450
11.3 Network in Network（NiN） 460
11.4 マルチブランチネットワーク（GoogLeNet） 470
11.5 バッチ正規化 483
11.6 残差ネットワーク（ResNet）と ResNeXt 504
11.7 密に接続されたネットワーク（DenseNet） 525
11.8 畳み込みネットワークのアーキテクチャ設計 539

12 再帰ニューラルネットワーク 552
12.1 シーケンスの扱い 554
12.2 生テキストを系列データに変換する 565
12.3 言語モデル 572
12.4 再帰型ニューラルネットワーク 578
12.5 ゼロからの再帰ニューラルネットワークの実装 583
12.6 リカレントニューラルネットワークの簡潔な実装 598
12.7 時間を通した逆伝播 602

13 現代的な再帰型ニューラルネットワーク 609
13.1 長短期記憶（LSTM） 610
13.2 ゲート付き再帰ユニット（GRU） 622
13.3 深層再帰ニューラルネットワーク 632
13.4 双方向リカレントニューラルネットワーク 638
13.5 機械翻訳とデータセット 642
13.6 エンコーダ–デコーダアーキテクチャ 648
13.7 機械翻訳のためのシーケンス・ツー・シーケンス学習 653
13.8 ビームサーチ 675

14 注意機構と Transformer 680
14.1 クエリ、キー、値 682
14.2 類似度によるアテンションプーリング 686
14.3 アテンションのスコアリング関数 692
14.4 バーダナウ注意機構 705
14.5 マルチヘッドアテンション 722
14.6 自己注意機構と位置エンコーディング 729
14.7 Transformer アーキテクチャ 738
14.8 画像向け Transformer 764
14.9 Transformer による大規模事前学習 772

15 最適化アルゴリズム 784
15.1 最適化と深層学習 784

15.2 凸性 794
15.3 勾配降下法 804
15.4 確率的勾配降下法 824
15.5 ミニバッチ確率的勾配降下法 834
15.6 モメンタム 852
15.7 Adagrad 869
15.8 RMSProp 879
15.9 Adadelta 886
15.10 Adam 891
15.11 学習率スケジューリング 898

16 計算性能 917
16.1 コンパイラとインタプリタ 917
16.2 非同期計算 925
16.3 自動並列化 929
16.4 ハードウェア 934
16.5 複数 GPU での学習 948
16.6 複数 GPU のための簡潔な実装 961
16.7 パラメータサーバー 966

17 コンピュータビジョン 975
17.1 画像拡張 976
17.2 ファインチューニング 989
17.3 物体検出とバウンディングボックス 998
17.4 アンカーボックス 1004
17.5 マルチスケール物体検出 1027
17.6 物体検出データセット 1034
17.7 シングルショット・マルチボックス検出 1039
17.8 Region-based CNNs (R-CNNs) 1056
17.9 セマンティックセグメンテーションとデータセット 1063
17.10 転置畳み込み 1074
17.11 完全畳み込みネットワーク 1081
17.12 ニューラル・スタイル変換 1092
17.13 Kaggle における画像分類（CIFAR-10） 1104
17.14 Kaggle における犬種識別（ImageNet Dogs） 1119

18 自然言語処理: 事前学習 1132
18.1 単語埋め込み（word2vec） 1133
18.2 近似学習 1138
18.3 単語埋め込みの事前学習のためのデータセット 1141
18.4 word2vec の事前学習 1158
18.5 グローバルベクトル（GloVe）による単語埋め込み 1165
18.6 サブワード埋め込み 1169
18.7 単語の類似性とアナロジー 1174
18.8 Transformer による双方向エンコーダ表現（BERT） 1186

18.9 BERT の事前学習のためのデータセット 1198
18.10 BERT の事前学習 1211

19 自然言語処理: 応用 1219
19.1 感情分析とデータセット 1220
19.2 感情分析: 再帰型ニューラルネットワークの使用 1227
19.3 畳み込みニューラルネットワークを用いた感情分析 1233
19.4 自然言語推論とデータセット 1243
19.5 自然言語推論: Attention の利用 1251
19.6 系列レベルおよびトークンレベルのアプリケーションのための BERT のファインチューニング 1261
19.7 自然言語推論: BERT のファインチューニング 1266

20 強化学習 1275
20.1 マルコフ決定過程（MDP） 1276
20.2 値反復 1279
20.3 Q 学習 1285

21 ガウス過程 1291
21.1 ガウス過程入門 1292
21.2 ガウス過程の事前分布 1301
21.3 ガウス過程推論 1307

22 ハイパーパラメータ最適化 1320
22.1 ハイパーパラメータ最適化とは何か？ 1320
22.2 ハイパーパラメータ最適化 API 1329
22.3 非同期ランダムサーチ 1337
22.4 マルチフィデリティ・ハイパーパラメータ最適化 1348
22.5 非同期逐次半減法 1364

23 生成的敵対的ネットワーク 1379
23.1 生成的敵対的ネットワーク 1379
23.2 Deep Convolutional Generative Adversarial Networks 1389

24 レコメンデーションシステム 1406
24.1 レコメンダーシステムの概要 1406
24.2 MovieLens データセット 1409
24.3 行列分解 1413
24.4 AutoRec: オートエンコーダによるレーティング予測 1418
24.5 レコメンダシステムのための個人化ランキング 1422
24.6 パーソナライズドランキングのためのニューラル協調フィルタリング 1424
24.7 シーケンスを考慮した推薦システム 1431
24.8 特徴豊富な推薦システム 1437
24.9 因子分解機 1440

24.10 Deep Factorization Machines 1443

25 付録: 深層学習のための数学 1448
25.1 幾何と線形代数的操作 1449
25.2 固有分解 1470
25.3 単変数微積分 1485
25.4 多変数微積分 1502
25.5 積分計算 1523
25.6 確率変数 1537
25.7 最尤推定 1563
25.8 分布 1570
25.9 ナイーブベイズ 1600
25.10 統計学 1613
25.11 情報理論 1628

26 付録: ディープラーニングのためのツール 1649
26.1 Jupyter Notebook の使用 1649
26.2 Amazon SageMaker の使用 1654
26.3 AWS EC2 インスタンスの使用 1657
26.4 Google Colab を使う 1666
26.5 サーバーと GPU の選択 1667
26.6 この本への貢献 1671
26.7 ユーティリティ関数とクラス 1676
26.8 d2l API ドキュメント 1706

Bibliography 1709

1 序文

ほんの数年前まで、大企業やスタートアップで知的な製品やサービスを開発する深層学習研究者の大軍勢は存在しなかった。私たちがこの分野に入った当時、機械学習は日刊紙の見出しを飾るようなものではなかった。私たちの親は機械学習が何であるかをまったく知らず、ましてやなぜ医学や法学といった王道のキャリアではなく、あえてこの分野を選ぶのか理解していなかった。当時の機械学習は、将来性は期待されていたものの、主として学術的な分野に留まっており、産業界での重要性は音声認識やコンピュータビジョンなど、ごく限られた実世界への応用に限られていた。さらに、これらの応用の多くは膨大なドメイン知識を必要としたため、機械学習は全体の一部にすぎず、（特定の応用領域ごとに）まったく別個の分野だとみなされることもしばしばであった。また、本書で扱う深層学習手法の前身であるニューラルネットワークは、当時、一般に時代遅れだと考えられていた。

しかし、わずか数年のうちに、深層学習は世界を驚かせ、コンピュータビジョン、自然言語処理、自動音声認識、強化学習、生体医療情報学といった多岐にわたる分野で急速な進歩を牽引してきた。さらに、実用上極めて重要な多くの課題で深層学習が成功したことは、理論機械学習や統計学の発展をも促した。こうした進歩を手にした今では、以前よりもはるかに高い自律性を備えた自動運転車（もっとも、いくつかの企業が主張するほどではないが）や、曖昧な点を質問してコードをデバッグする対話システム、そしてかつては何十年も先のことだと思われていた、囲碁のようなボードゲームで世界最高の人間プレイヤーを打ち負かすソフトウェアエージェントを構築できるようになった。また、これらのツールは産業や社会にますます広範な影響を及ぼしており、映画の制作方法や病気の診断方法を根本から変えつつあり、天体物理学から気候モデリング、気象予測、生物医学に至る基礎科学の分野でも重要な役割を果たしつつある。

1.1 本書について

本書は、深層学習を身近なものにするための私たちの試みであり、概念、文脈、そしてコードを教えることを目的としている。

1.1.1 コード、数式、HTML を一体化した媒体

あらゆる計算技術がその真価を発揮するためには、十分に理解され、十分に文書化され、成熟して保守の行き届いたツールによって支えられていなければならない。重要なアイデアは明確に要約され、新しい実践者が最新の知識に追いつくために必要な導入時間を最小限に抑えるべきである。成熟したライブラリは一般的な作業を自動化し、模範的なコードは実践者が一般的な応用を自分のニーズに合わせて修正・適用・拡張しやすくするべきである。

例として、動的な Web アプリケーションを考えてみよう。1990 年代には、Amazon のような多くの企業がデータベース駆動型の Web アプリケーションを成功裏に開発していたが、この技術が創造的な起業家を支援する潜在能力を本格的に開花させたのは、強力で文書化の行き届いたフレームワークの発展にも後押しされた、ここ 10 年ほどのことだった。

深層学習の可能性を検証することには独特の難しさがある。なぜなら、単一の応用であってもさまざまな分野の知識が結びつくからである。深層学習を適用するには、同時に次のことを理解する必要がある。(i) 問題を特定の形で定式化する動機 (ii) 与えられたモデルの数学的形式 (iii) モデルをデータに適合させるための最適化アルゴリズム (iv) モデルが未知データに一般化すると期待できる条件を教えてくれる統計的原理と、実際に一般化したことを検証するための実践的手法 (v) モデルを効率よく学習させるために必要な工学的技法、すなわち数値計算の落とし穴を回避し、利用可能なハードウェアを最大限に活用する方法

問題を定式化するために必要な批判的思考力、それを解くための数学、そしてその解を実装するためのソフトウェアツールを、すべて一か所で教えることは、非常に大きな課題である。本書の目標は、将来の実践者を迅速に実務レベルへと引き上げるための、統一的なリソースを提示することである。

本書の企画を始めたとき、次の条件を同時に満たすリソースは存在しなかった。(i) 常に最新であること (ii) 現代の機械学習の実践の広がりを十分な技術的深さでカバーしていること (iii) 教科書に期待される質の高い解説と、実践的チュートリアルに期待される洗練された実行可能コードを交互に提示していること

私たちは、特定の深層学習フレームワークの使い方を示すコード例（たとえば、TensorFlow で行列を使った基本的な計算を行う方法）や、特定の技法を実装するためのコード片（たとえば、LeNet、AlexNet、ResNet などのコードスニペット）が、さまざまなブログ記事や GitHub リポジトリに散在しているのを数多く見つけた。しかし、これらの例は通常、ある手法をどのように実装するかに焦点を当てており、なぜそのようなアルゴリズム上の判断がなされたのか、という議論に欠けていた。特定の話題を掘り下げるための、たとえばウェブサイト Distill[1] に掲載された魅力的な記事や個人ブログのような対話的なリソースが散発的に現れることもあったが、それらは深層学習の限られた話題しか扱っておらず、関連するコードが伴わないことも多くあった。一方で、いくつかの深層学習の教科書　たとえば、深層学習の基礎を包括的に概説した Goodfellow *et al.* (2016) など　も登場していたが、これらのリソースは概念の説明とコードによる実装を結びつけておらず、読者がいざ実装しようとしたときに途方に暮れ

[1] 

てしまうこともあった。さらに、多くの優れたリソースが商業的な講座提供者の有料の壁（ペイウォール）の向こうに隠されている。

私たちは、次の条件を満たすリソースを作ろうとした。(i) 誰もが無料で利用できること (ii) 実際に応用機械学習研究者となるための出発点として、十分な技術的深さを備えていること (iii) 実行可能なコードを含み、実践で問題をどのように解くかを読者に示すこと (iv) 私たち自身だけでなく、広いコミュニティを通じても迅速に更新できること

これらの目標は、しばしば互いに衝突した。数式、定理、引用は LaTeX で管理・整形するのが最適である。コードは Python で記述するのが最適である。そして Web ページは HTML と JavaScript が本来の形式である。さらに、読者がこれらの内容を、実行可能なコードとしても、紙の書籍としても、ダウンロード可能な PDF としても、そしてインターネット上の Web サイトとしても利用できるようにしたいと考えた。こうした要求をすべて満たす既存のワークフローは見当たらなかったため、私たちは独自にシステムを組み立てることにした（Section 26.6）。私たちは、ソースコードを共有しコミュニティの貢献を促進するために GitHub を採用し、コード・数式・テキストを混在させるために Jupyter notebook を採用し、レンダリングエンジンとして Sphinx を採用し、議論の場として Discourse を採用した。私たちのシステムは完璧ではないが、これらの選択は相反する要求の間で妥協点を見いだすものである。私たちは、*Dive into Deep Learning* が、このような統合されたワークフローを用いて出版された最初の書籍かもしれないと自負している。

1.1.2 実践しながら学ぶ

多くの教科書は、概念を順に提示し、それぞれを徹底的に詳述する構成をとる。たとえば、Bishop (2006) の優れた教科書は、各トピックを非常に丁寧に教えるため、線形回帰の章にたどり着くまでにかなりの労力を要する。専門家はその徹底ぶりをまさに理由としてこの教科書を高く評価するが、真の初心者にとっては、この性質が入門書としての参入障壁を高めることになる。

本書では、ほとんどの概念を必要なときにその場で教える手法をとる。言い換えれば、何らかの実用的な目的を達成するために必要になった瞬間に、その概念を学ぶことになる。冒頭では線形代数や確率といった基礎的な前提知識を学ぶために少し時間を割くが、より難解な概念を気にする前に、まずは最初のモデルを学習させる満足感を読者にぜひ味わってほしいのである。

線形代数や確率の基礎を短期集中で学ぶためのいくつかの予備ノートブックを除けば、その後の各章は、適度な数の新しい概念を導入すると同時に、実データセットを用いた、自己完結した実行可能なコード例を提供する。これは構成上の課題を生んだ。いくつかのモデルは、論理的には一つのノートブックにまとめることができる。また、いくつかのアイデアは、複数のモデルを順番に実行しながら教えるのが最適かもしれない。それに対して、私たちがとった一つの動作例につき一つのノートブックという方針には、大きな利点がある。それは、読者が私たちのコードを活用して自分自身の研究プロジェクトを始めるこ

とを、できる限り簡単にしてくれるからである。ノートブックをコピーして、自分の目的に合わせて修正を始めるだけでよいのである。

本書全体を通して、必要に応じて実行可能なコードと背景説明を交互に配置している。一般的に、私たちはツールを完全に説明する前に、まず使ってみて実感できる形にしている（より詳細な背景説明は後から補うことが多い）。たとえば、確率的勾配降下法を、その数学的な裏付けや直感的な仕組みを説明する前に実践で使うことがある。これは、実践者が問題を素早く解くための強力な武器をいち早く手に入れる助けになるが、その代わりに、いくつかの学習上の「順序の逆転」を受け入れてもらう必要を生じさせる。

本書は深層学習の概念をゼロから教える。ときには、現代の深層学習フレームワークでは通常ユーザーから隠されているような、モデルの細部にまで踏み込む。これは特に基礎的なチュートリアルで顕著であり、それは読者に、ある層やオプティマイザの内部で何が起きているのかをすべて理解してほしいからである。こうした場合、私たちはしばしばコード例を二つのバージョンで示す。一つは、NumPy に似た機能と自動微分だけに頼り、すべてをゼロから実装するバージョンである。もう一つは、深層学習フレームワークの高水準 API を使って簡潔なコードを書く、より実践的なバージョンである。あるコンポーネントの動作原理をコードで説明した後は、その後のチュートリアルでは高水準 API による実装に頼ることになる。

1.1.3 内容と構成

本書は、おおむね三つの部分に分けられる。すなわち、前提知識、深層学習技法、そして実システムと応用に焦点を当てた高度な話題である（Fig. 1.1.1）。

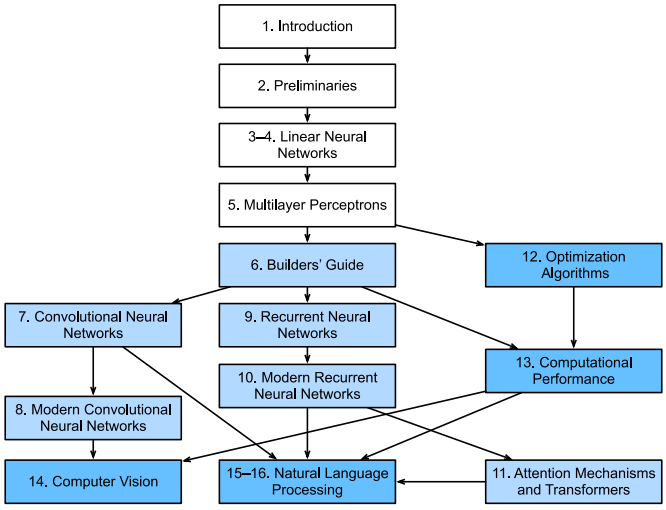


Fig. 1.1.1 Book structure.

- **第 1 部: 基礎と前提知識**。Section 4 は深層学習の入門である。次に、Section

5 では、データの保存と操作の方法や、線形代数・微積分・確率の初歩的な概念に基づくさまざまな数値演算の適用方法など、実践的な深層学習に必要な前提知識を素早く身につける。Section 6 と Section 8 では、回帰と分類、線形モデル、多層パーセプトロン、過学習と正則化を含む、深層学習における最も基本的な概念と技法を扱う。

- **第 2 部: 現代的な深層学習技法**。Section 9 では、深層学習システムの主要な計算要素を説明し、より複雑なモデルを後続で実装するための基盤を築く。続いて、Section 10 と Section 11 では、現代のコンピュータビジョンシステムの大半の中核を成す強力なツールである畳み込みニューラルネットワーク（CNN）を紹介する。同様に、Section 12 と Section 13 では、データの逐次的な（たとえば時間的な）構造を活用し、自然言語処理や時系列予測で一般的に用いられる再帰型ニューラルネットワーク（RNN）を紹介する。Section 14 では、いわゆる注意機構に基づく比較的新しいモデル群を説明する。これは、ほとんどの自然言語処理タスクにおいて RNN に取って代わった支配的なアーキテクチャである。これらの節を通じて、深層学習実践者が広く用いている、最も強力で汎用的なツールを身につけることができる。

- **第 3 部: スケーラビリティ、効率性、応用**（オンライン[2] で利用可能）。第 12 章では、深層学習モデルの学習に用いられるいくつかの一般的な最適化アルゴリズムを議論する。次に、第 13 章では、深層学習コードの計算性能に影響を与えるいくつかの重要な要因を検討する。続いて、第 14 章では、コンピュータビジョンにおける深層学習の主要な応用を示す。最後に、第 15 章と第 16 章では、言語表現モデルを事前学習し、それを自然言語処理タスクに適用する方法を示す。

[2] 

1.1.4 コード

本書の大部分の節には実行可能なコードが含まれている。私たちは、試行錯誤を通じてコードを少しずつ調整し、その結果を観察することで、より良い直感が育まれると考えている。理想的には、洗練された数学理論が、望ましい結果を得るためにコードをどのように調整すべきかを正確に教えてくれるはずである。しかし、今日の深層学習実践者は、確かな理論による指針がない領域を手探りで進まざるを得ないことがしばしばある。私たちが最善を尽くしても、さまざまな技法がなぜ有効なのかという形式的な説明は、依然として多くの理由から不足している。それらの理由は、これらのモデルを特徴づける数学が非常に難しいこと、説明が現在まだ明確に定義されていないデータの性質に依存している可能性があること、そしてこの分野への本格的な研究がようやく本格化したばかりであること、などである。深層学習理論が進歩するにつれて、本書の将来の版では、現在利用可能な知識を凌駕するような深い洞察を提供できることを期待している。

不要な繰り返しを避けるため、私たちは最も頻繁にインポートして使用する関数やクラスのいくつかを `d2l` パッケージにまとめている。全体を通して、コードブロック（関数、クラス、あるいは import 文の集合など）には `#@save` を付け、それらが後で `d2l` パッケージ経由で参照されることを示す。これらのクラ

スと関数の詳細な概要は Section 26.8 で説明する。d2l パッケージは軽量で、以下の依存関係だけを必要とする。

```
#@save
import inspect
import collections
from collections import defaultdict
from IPython import display
import math
from matplotlib import pyplot as plt
from matplotlib_inline import backend_inline
import os
import pandas as pd
import random
import re
import shutil
import sys
import tarfile
import time
import requests
import zipfile
import hashlib
d2l = sys.modules[__name__]
```

本書のコードの大部分は PyTorch を基盤にしている。PyTorch は、深層学習研究コミュニティに熱心に受け入れられている人気のオープンソースフレームワークである。本書のすべてのコードは、最新の安定版 PyTorch でテストを通過している。しかし、深層学習の急速な発展により、印刷版の一部のコードは将来の PyTorch バージョンでは正しく動作しない可能性がある。オンライン版は最新の状態に保つ予定である。問題が発生した場合は、インストール (page 19) を参照して、コードと実行環境を更新してほしい。以下に PyTorch 実装での依存関係を示す。

```
#@save
import numpy as np
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from torchvision import transforms
from PIL import Image
from scipy.spatial import distance_matrix
```

```
#@save
from mxnet import autograd, context, gluon, image, init, np, npx
from mxnet.gluon import nn, rnn
```

```
#@save
from dataclasses import field
from functools import partial
import flax
```

(continues on next page)

(continued from previous page)

```
from flax import linen as nn
from flax.training import train_state
import jax
from jax import numpy as jnp
from jax import grad, vmap
import numpy as np
import optax
import tensorflow as tf
import tensorflow_datasets as tfds
from types import FunctionType
from typing import Any
```

```
#@save
import numpy as np
import tensorflow as tf
```

1.1.5 対象読者

本書は、深層学習の実践的技法をしっかり理解したい学生（学部生・大学院生）、エンジニア、研究者を対象としている。すべての概念をゼロから説明するため、読者に深層学習や機械学習の事前知識は求めない。しかし、深層学習の手法について十分に説明するには、ある程度の数学とプログラミングの知識が必要である。そのため本書では、線形代数、微積分、確率論、および Python プログラミングの基礎的な知識があることだけを前提とする。万一何か忘れてしまった場合でも、本書に登場する数学の大半については、オンライン付録[3] が復習用の解説を提供している。また、私たちは一部の例外を除き、数学的な厳密さよりも直感とアイデアを優先している。さらに深く数学的知識を広げたい読者のために、本書以外にも優れた参考資料をおすすめしている。Bollobás (1999) の *Linear Analysis* は、線形代数と関数解析を非常に深く扱っている。*All of Statistics* (Wasserman, 2013) は、統計学への素晴らしい入門書である。Joe Blitzstein 氏による確率と推論に関する 書籍[4] と 講義[5] は、教育的価値の高い素晴らしいリソースである。また、これまで Python を使ったことがない読者には、こちらの Python チュートリアル[6] を一読することをお勧めする。

1.1.6 ノートブック、Web サイト、GitHub、フォーラム

すべてのノートブックは D2L.ai の Web サイト[7] と GitHub[8] からダウンロードできる。本書のどの節についても質問があるときは、各ノートブックの末尾にある関連議論ページへのリンクをたどることができる。

1.2 謝辞

英語版と中国語版の草稿の両方に対して、何百人もの貢献者に深く感謝する。彼らは内容の改善に協力し、貴重なフィードバックを提供してくれた。本書は当初、MXNet を主要フレームワークとして実装された。以前の MXNet コードの大部分を、それぞれ PyTorch と TensorFlow の実装へ適応させてくれた Anirudh Dagar と Yuan Tang に感謝する。2021 年 7 月以降、私たちは本書を PyTorch、MXNet、TensorFlow で再設計・再実装し、PyTorch を主要フレームワークとして採用した。より最近の PyTorch コードの大部分を JAX 実装へ適応させてくれた Anirudh Dagar に感謝する。中国語版草稿において、より最近の PyTorch コードの大部分を PaddlePaddle 実装へ適応させてくれた Baidu の Gaosheng Wu、Liujun Hu、Ge Zhang、Jiehang Xie に感謝する。PDF のビルドにおいて、出版社の LaTeX スタイルを統合してくれた Shuai Zhang に感謝する。

GitHub では、この英語版草稿をより良いものにしてくれたすべての貢献者に感謝する。彼らの GitHub ID または名前は（順不同）次のとおりである。alxnorden, avinashingit, bowen0701, brettkoonce, Chaitanya Prakash Bapat, cryptonaut, Davide Fiocco, edgarroman, gkutiel, John Mitro, Liang Pu, Rahul Agarwal, Mohamed Ali Jamaoui, Michael (Stu) Stewart, Mike Müller, NRauschmayr, Prakhar Srivastav, sad-, sfermigier, Sheng Zha, sundeepteki, topecongiro, tpdi, vermicelli, Vishaal Kapoor, Vishwesh Ravi Shrimali, YaYaB, Yuhong Chen, Evgeniy Smirnov, lgov, Simon Corston-Oliver, Igor Dzreyev, Ha Nguyen, pmuens, Andrei Lukovenko, senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, Bowen Li, Aarush Ahuja, Prasanth Buddareddygari, brianhendee, mani2106, mtn, lkevinzc, caojilin, Lakshya, Fiete Lüer, Surbhi Vijayvargeeya, Muhyun Kim, dennismalmgren, adursun, Anirudh Dagar, liqingnz, Pedro Larroy, lgov, ati-ozgur, Jun Wu, Matthias Blume, Lin Yuan, geogunow, Josh Gardner, Maximilian Böther, Rakib Islam, Leonard Lausen, Abhinav Upadhyay, rongruosong, Steve Sedlmeyer, Ruslan Baratov, Rafael Schlatter, liusy182, Giannis Pappas, ati-ozgur, qbaza, dchoi77, Adam Gerson, Phuc Le, Mark Atwood, christabella, vn09, Haibin Lin, jjangga0214, RichyChen, noelo, hansent, Giel Dops, dvincent1337, WhiteD3vil, Peter Kulits, codypenta, joseppinilla, ahmaurya, karolszk, heytitle, Peter Goetz, rigtorp, Tiep Vu, sfilip, mlxd, Kale-ab Tessera, Sanjar Adilov, MatteoFerrara, hsneto, Katarzyna Biesialska, Gregory Bruss, Duy−Thanh Doan, paulaurel, graytowne, Duc Pham, sl7423, Jaedong Hwang, Yida Wang, cys4, clhm, Jean Kaddour, austinmw, trebeljahr, tbaums, Cuong V. Nguyen, pavelkomarov, vzlamal, NotAnotherSystem, J-Arun-Mani, jancio, eldarkurtic, the-great-shazbot, doctorcolossus, gducharme, cclauss, Daniel-Mietchen, hoonose, biagiom, abhinavsp0730, jonathanhrandall, ysraell, Nodar Okroshiashvili, UgurKap, Jiyang Kang, StevenJokes, Tomer Kaftan, liweiwp, netyster, ypandya, NishantTharani, heiligerl, SportsTHU, Hoa Nguyen, manuel-arno-korfmann-webentwicklung, aterzis-personal, nxby, Xiaoting He, Josiah Yoder, mathresearch, mzz2017, jroberayalas, iluu, ghejc, BSharmi, vkramdev, simonwardjones, LakshKD, TalNeoran, djliden, Nikhil95, Oren Barkan, guoweis, haozhu233, pratikhack, Yue Ying, tayfununal, steinsag, charleybeller, Andrew Lumsdaine, Jiekui Zhang, Deepak Pathak, Florian Donhauser,

Tim Gates, Adriaan Tijsseling, Ron Medina, Gaurav Saha, Murat Semerci, Lei Mao, Levi McClenny, Joshua Broyde, jake221, jonbally, zyhazwraith, Brian Pulfer, Nick Tomasino, Lefan Zhang, Hongshen Yang, Vinney Cavallo, yuntai, Yuanxiang Zhu, amarazov, pasricha, Ben Greenawald, Shivam Upadhyay, Quanshangze Du, Biswajit Sahoo, Parthe Pandit, Ishan Kumar, HomunculusK, Lane Schwartz, varadgunjal, Jason Wiener, Armin Gholampoor, Shreshtha13, eigen-arnav, Hyeonggyu Kim, EmilyOng, Bálint Mucsányi, Chase DuBois, Juntian Tao, Wenxiang Xu, Lifu Huang, filevich, quake2005, nils-werner, Yiming Li, Marsel Khisamutdinov, Francesco "Fuma"Fumagalli, Peilin Sun, Vincent Gurgul, qingfengtommy, Janmey Shukla, Mo Shan, Kaan Sancak, regob, AlexSauer, Gopalakrishna Ramachandra, Tobias Uelwer, Chao Wang, Tian Cao, Nicolas Corthorn, akash5474, kxxt, zxydi1992, Jacob Britton, Shuangchi He, zhmou, krahets, Jie-Han Chen, Atishay Garg, Marcel Flygare, adtygan, Nik Vaessen, bolded, Louis Schlessinger, Balaji Varatharajan, atgctg, Kaixin Li, Victor Barbaros, Riccardo Musto, Elizabeth Ho, azimjonn, Guilherme Miotto, Alessandro Finamore, Joji Joseph, Anthony Biel, Zeming Zhao, shjustinbaek, gab-chen, nantekoto, Yutaro Nishiyama, Oren Amsalem, Tian-MaoMao, Amin Allahyar, Gijs van Tulder, Mikhail Berkov, iamorphen, Matthew Caseres, Andrew Walsh, pggPL, RohanKarthikeyan, Ryan Choi, and Likun Lei.

本書の執筆にあたり、Amazon Web Services、特に Wen-Ming Ye、George Karypis、Swami Sivasubramanian、Peter DeSantis、Adam Selipsky、そして Andrew Jassy の寛大な支援に感謝する。十分な時間、リソース、同僚との議論、そして彼らからの絶え間ない励ましがなければ、この本は実現しなかったであろう。出版準備の過程では、Cambridge University Press が素晴らしい支援を提供してくれた。私たちは、委嘱編集者 David Tranah の助力とプロフェッショナリズムに深く感謝する。

1.3 要約

深層学習はパターン認識に革命をもたらし、コンピュータビジョン、自然言語処理、自動音声認識といった多様な分野で、現在さまざまな技術を支える基盤となっている。深層学習をうまく適用するには、問題をどのように定式化するか、モデリングの基本数学、モデルをデータに適合させるアルゴリズム、そしてそれらすべてを実装する工学的技法を理解する必要がある。本書は、文章、図、数式、コードをすべて一か所にまとめた包括的なリソースを提供する。

1.4 演習

1. 自分のコンピュータに Python をインストールすること。

2. 節の下部にあるリンクからフォーラムへ移動し、著者やより広いコミュニティと交流しながら、助けを求めたり、本書について議論したり、質問への答えを見つけたりしてほしい。

2 インストール

本書を使い始めるには、Python を実行するための環境、Jupyter Notebook、関連ライブラリ、そして本書のコードを実行するために必要なパッケージを準備する必要がある。

2.1 Miniconda のインストール



9 最も簡単な方法は Miniconda[9] をインストールすることである。Python 3.x が必要であることに注意してほしい。すでにマシンに conda がインストールされている場合は、以下の手順を省略してよい。

Miniconda の Web サイトにアクセスし、Python 3.x のバージョンとマシンのアーキテクチャに合わせて、システムに適したものを選ぶ。ここでは Python のバージョンを 3.9（動作確認済みのバージョン）と想定する。macOS を使用している場合は、名前に“MacOSX”という文字列を含む bash スクリプトをダウンロードし、ダウンロード先のディレクトリに移動して、以下のようにインストールを実行する（Intel Mac を例とする）：

```
# ファイル名はバージョンにより異なる場合がある
sh Miniconda3-py39_4.12.0-MacOSX-x86_64.sh -b
```

Linux ユーザーは、名前に“Linux”という文字列を含むファイルをダウンロードし、ダウンロード先で以下を実行する：

```
# ファイル名はバージョンにより異なる場合がある
sh Miniconda3-py39_4.12.0-Linux-x86_64.sh -b
```



10 Windows ユーザーは、オンラインの手順[10] に従って Miniconda をダウンロードし、インストールしてほしい。Windows では、cmd を検索してコマンドプロンプト（コマンドラインインタプリタ）を開き、コマンドを実行できる。

次に、conda を直接実行できるようにシェルを初期化する。

```
~/miniconda3/bin/conda init
```

その後、現在のシェルを閉じて再度開く。以下のようにして新しい環境を作成できるはずである：

```
conda create --name d2l python=3.9 -y
```

これで d2l 環境を有効化できる：

```
conda activate d2l
```

2.2 ディープラーニングフレームワークと d2l パッケージのインストール

いずれかのディープラーニングフレームワークをインストールする前に、まずマシンに適切な GPU があるかどうかを確認してほしい（標準的なノートパソコンでディスプレイを駆動する GPU は、ここでの目的には関係ない）。例えば、コンピュータに NVIDIA GPU があり、CUDA[11] がインストールされているなら、準備は整っている。マシンに GPU が搭載されていなくても、今のところ心配する必要はない。最初の数章を進めるには、CPU だけでも十分な性能がある。ただし、より大きなモデルを実行する前には、GPU にアクセスできるようにしておきたいことを覚えておいてほしい。

11 

PyTorch は、CPU 対応版または GPU 対応版のいずれでも、以下のようにインストールできる（指定したバージョンは執筆時点で検証済みである）：

```
pip install torch==2.0.0 torchvision==0.15.1
```

次のステップは、本書全体で頻繁に使用する関数やクラスをまとめるために開発された d2l パッケージのインストールである：

```
pip install d2l==1.0.3
```

2.3 コードのダウンロードと実行

次に、各章のコードブロックを実行できるよう、ノートブックをダウンロードする。D2L-JP の Web サイト[12] 上部のナビゲーションバーにある「ノートブック」リンクからコードをダウンロードし、解凍してほしい。あるいは、コマンドラインから以下のようにノートブックを取得することもできる：

12 

```
mkdir d2l-jp && cd d2l-jp
curl https://d2l-jp.me/d2l-jp.zip -o d2l-jp.zip
unzip d2l-jp.zip && rm d2l-jp.zip
cd pytorch
```

unzip がまだインストールされていない場合は、まず sudo apt-get install unzip を実行する。これで、以下を実行して Jupyter Notebook サーバーを起動できる：

```
jupyter notebook
```

この時点で、Web ブラウザから http://localhost:8888 （すでに自動的に開いている場合もある）にアクセスできるようになる。そこから、本書の各節のコードを実行できる。新しいコマンドラインウィンドウを開くたびに、D2L のノートブックを実行する前、またはパッケージ（ディープラーニングフレームワークあるいは d2l パッケージ）を更新する前に、conda activate d2l を実行して実行環境を有効化する必要がある。環境を終了するには、conda deactivate を実行する。

3 記法

本書全体を通して、以下の記法規則に従います。
これらの記号の一部はプレースホルダであり、他のものは特定の対象を指します。
一般的な目安として、不定冠詞の“a”はしばしば、その記号がプレースホルダであり、同じ形式の記号が同種の別の対象を表しうることを示します。
たとえば、“x: a scalar”は、小文字の文字が一般にスカラー値を表すことを意味しますが、“$\mathbb{Z}$: the set of integers”は記号 $\mathbb{Z}$ を特定して指しています。

3.1 数値オブジェクト

- x: スカラー
- $\mathbf{x}$: ベクトル
- $\mathbf{X}$: 行列
- X: 一般のテンソル
- $\mathbf{I}$: 単位行列（ある与えられた次元のもの）、すなわち対角成分がすべて 1、非対角成分がすべて 0 の正方行列
- x_i, $[\mathbf{x}]_i$: ベクトル $\mathbf{x}$ の i^{th} 要素
- x_{ij}, $x_{i,j}$,$[\mathbf{X}]_{ij}$, $[\mathbf{X}]_{i,j}$: 行列 $\mathbf{X}$ の第 i 行第 j 列の要素

3.2 集合論

- $\mathcal{X}$: 集合
- $\mathbb{Z}$: 整数全体の集合
- $\mathbb{Z}^+$: 正の整数全体の集合

- $\mathbb{R}$: 実数全体の集合
- $\mathbb{R}^n$: 実数からなる n 次元ベクトル全体の集合
- $\mathbb{R}^{a\times b}$: a 行 b 列の実数行列全体の集合
- $|\mathcal{X}|$: 集合 $\mathcal{X}$ の濃度（要素数）
- $\mathcal{A}\cup\mathcal{B}$: 集合 $\mathcal{A}$ と $\mathcal{B}$ の和集合
- $\mathcal{A}\cap\mathcal{B}$: 集合 $\mathcal{A}$ と $\mathcal{B}$ の共通部分
- $\mathcal{A}\setminus\mathcal{B}$: $\mathcal{A}$ から $\mathcal{B}$ を引いた差集合（$\mathcal{A}$ のうち $\mathcal{B}$ に属さない要素のみを含む）

3.3 関数と演算子

- $f(\cdot)$: 関数
- $\log(\cdot)$: 自然対数（底 e）
- $\log_2(\cdot)$: 底 2 の対数
- $\exp(\cdot)$: 指数関数
- $\mathbf{1}(\cdot)$: 指示関数。ブール値の引数が真なら 1、それ以外なら 0 を返す
- $\mathbf{1}_{\mathcal{X}}(z)$: 集合所属指示関数。要素 z が集合 $\mathcal{X}$ に属すれば 1、それ以外なら 0 を返す
- $(\cdot)^\top$: ベクトルまたは行列の転置
- $\mathbf{X}^{-1}$: 行列 $\mathbf{X}$ の逆行列
- $\odot$: ハダマード積（要素ごとの積）
- $[\cdot,\cdot]$: 連結
- $\|\cdot\|_p$: ℓ_p ノルム
- $\|\cdot\|$: ℓ_2 ノルム
- $\langle\mathbf{x},\mathbf{y}\rangle$: ベクトル $\mathbf{x}$ と $\mathbf{y}$ の内積（ドット積）
- $\sum$: 要素の集合に対する総和
- $\prod$: 要素の集合に対する総乗
- $\stackrel{\text{def}}{=}$: 左辺の記号の定義として主張される等式

3.4 微積分

- $\frac{dy}{dx}$: x に関する y の導関数
- $\frac{\partial y}{\partial x}$: x に関する y の偏導関数
- $\nabla_{\mathbf{x}} y$: $\mathbf{x}$ に関する y の勾配
- $\int_a^b f(x)\, dx$: x に関して a から b までの f の定積分
- $\int f(x)\, dx$: x に関する f の不定積分

3.5 確率と情報理論

- X: 確率変数
- P: 確率分布
- $X \sim P$: 確率変数 X は分布 P に従う
- $P(X = x)$: 確率変数 X が値 x をとる事象に割り当てられる確率
- $P(X \mid Y)$: Y が与えられたときの X の条件付き確率分布
- $p(\cdot)$: 分布 P に対応する確率密度関数（PDF）
- $E[X]$: 確率変数 X の期待値
- $X \perp Y$: 確率変数 X と Y は独立
- $X \perp Y \mid Z$: 確率変数 X と Y は、Z が与えられたとき条件付き独立
- σ_X: 確率変数 X の標準偏差
- $\mathrm{Var}(X)$: 確率変数 X の分散。σ_X^2 に等しい
- $\mathrm{Cov}(X, Y)$: 確率変数 X と Y の共分散
- $\rho(X, Y)$: X と Y のピアソン相関係数。$\frac{\mathrm{Cov}(X,Y)}{\sigma_X \sigma_Y}$ に等しい
- $H(X)$: 確率変数 X のエントロピー
- $D_{\mathrm{KL}}(P \| Q)$: 分布 Q から分布 P への KL ダイバージェンス（または相対エントロピー）

4 はじめに

最近まで、私たちが日常的に操作するコンピュータプログラムのほぼすべては、その振る舞いを正確に指定する厳密なルールの集合としてコーディングされていた。例えば、E コマースプラットフォームを管理するアプリケーションを作成したいとしよう。数時間ホワイトボードを囲んで問題について熟考した後、解決策の全体像として次のようなものに落ち着くかもしれない。(i) ユーザーは Web ブラウザやモバイルアプリで動作するインターフェースを通じてアプリケーションとやり取りする。(ii) アプリケーションは商用レベルのデータベースエンジンと連携し、各ユーザーの状態を追跡して過去の取引記録を保持する。(iii) アプリケーションの中枢では、ビジネスロジック（いわばアプリケーションの頭脳）が、考え得るあらゆる状況を、プログラムが取るべき対応するアクションにマッピングする一連のルールを規定する。

アプリケーションの頭脳を構築するために、プログラムが処理すべき一般的なイベントをすべて列挙するかもしれない。例えば、顧客が商品をショッピングカートに追加するボタンをクリックするたびに、プログラムはショッピングカートのデータベーステーブルにエントリを追加し、そのユーザーの ID とリクエストされた商品の ID を関連付ける。次に、考えられるすべてのコーナーケースを一つずつ確認し、ルールの妥当性をテストして必要な修正を加える試みを行うだろう。ユーザーが空のカートで決済を開始したらどうなるだろうか？最初の試みで完全に正しく動作させられる開発者はほとんどいないが（問題を解決するために何度かのテスト実行が必要かもしれない）、基本的には、実際の顧客を見る前にこのようなプログラムを書き上げ、自信を持って立ち上げることができる。機能する製品やシステムを駆動する自動化システムを手作業で設計する私たちの能力は、しばしば未知の状況においても発揮され、驚異的な認知的偉業である。そして、100% の確率で機能する解決策を考案できるのであれば、通常は機械学習について悩む必要はない。

成長を続ける機械学習の研究者コミュニティにとって幸いなことに、私たちが自動化したいと考えるタスクの多くは、人間の創意工夫に簡単には屈しない。あなたの知る最も優秀な頭脳を持つ人々とホワイトボードを囲んでいると想像してほしい。ただし、今回取り組むのは次のような問題のいずれかである。

- 地理情報、衛星画像、過去の一定期間の天候データを与えられ、明日の天候を予測するプログラムを作成する。
- 自由記述のテキストで表現された事実に基づく質問を受け取り、正確に回答するプログラムを作成する。

- 画像を与えられ、そこに描かれているすべての人物を特定し、それぞれの輪郭を描画するプログラムを作成する。
- ユーザーが気に入りそうだが、通常のブラウジングでは出会いそうにない商品を提示するプログラムを作成する。

これらの問題に対しては、エリートプログラマーでさえゼロから解決策をコーディングするのに苦労するだろう。その理由は様々である。求めるプログラムが時間とともに変化するパターンに従うため、固定された正解が存在しないこともある。そのような場合、成功する解決策は絶え間なく変化する世界に柔軟に適応しなければならない。また、関係性（例えばピクセルと抽象的なカテゴリとの間の関係）が複雑すぎて、数千から数百万回の計算を必要とし、未知の原理に従っている場合もある。画像認識の場合、タスクを実行するために必要な正確な手順は、私たちの意識的な理解を超えている。私たちの無意識な認知プロセスは簡単にタスクを実行しているにもかかわらずである。

機械学習（Machine learning）は、経験から学習できるアルゴリズムの研究である。機械学習アルゴリズムは、通常、観測データや環境との相互作用という形で経験を積むにつれて性能が向上する。これを、私たちが構築した決定論的な E コマースプラットフォームと比較してみよう。そのプラットフォームは、どれほど経験が蓄積されても、開発者自身が学習し、ソフトウェアを更新する時期だと判断するまでは、同じビジネスロジックに従い続ける。本書では、機械学習の基礎を教える。特にディープラーニング（Deep learning）に焦点を当てる。ディープラーニングは、コンピュータビジョン、自然言語処理、ヘルスケア、ゲノミクスなどの多様な分野でイノベーションを推進している強力な技術群である。

4.1 動機付けとなる例

執筆を始める前、この本の著者たちは、労働者の多くがそうであるように、カフェインを摂取しなければならなかった。私たちは車に乗り込み、運転を始めた。Alex は iPhone を使って「Hey Siri」と呼びかけ、電話の音声認識システムを起動した。次に Mu が「Blue Bottle コーヒーショップへの道順」と指示した。電話はすぐに彼のコマンドの書き起こしを表示した。また、私たちが道順を尋ねていることを認識し、リクエストを満たすためにマップアプリケーション（アプリ）を起動した。起動すると、マップアプリは複数のルートを特定した。各ルートの隣には、予測される移動時間が表示された。この話は教育上の便宜のために作られたものだが、スマートフォンとの日常的なやり取りが、わずか数秒の間に複数の機械学習モデルを稼働させる可能性があることを示している。

「Alexa」、「OK Google」、「Hey Siri」などのウェイクワード（wake word）に応答するプログラムを書くことだけを想像してみよう。Fig. 4.1.1 に示されているように、コンピュータとコードエディタだけが置かれた部屋で、一人でコー

ディングを試みてほしい。第一原理からそのようなプログラムをどのように書くのだろうか？考えてみてほしい…この問題は困難である。マイクは毎秒およそ 44,000 のサンプルを収集する。各サンプルは音波の振幅の測定値である。生の音声の短い断片を、その断片にウェイクワードが含まれているかどうかについての確信を持った予測 {yes, no} に、高い信頼性でマッピングできるルールとはどのようなものだろうか？行き詰まっても、心配する必要はない。私たちも、そのようなプログラムをゼロから書く方法は知らない。だからこそ、機械学習を使用するのである。

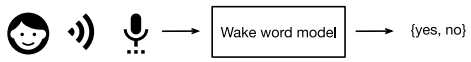


Fig. 4.1.1 ウェイクワードの特定

ここにトリックがある。入力を出力にマッピングする方法をコンピュータに明示的に指示する方法がわからなくても、私たち自身はその認知的偉業を実行できることがよくある。言い換えれば、「Alexa」という単語を認識するようにコンピュータをプログラミングする方法を知らなくても、あなた自身はそれを認識できるということだ。この能力を武器に、短い音声断片と関連するラベル（どの断片にウェイクワードが含まれているかを示す）の例を含む巨大なデータセット（dataset）を収集できる。現在主流となっている機械学習のアプローチでは、ウェイクワードを認識するシステムを明示的に設計しようとはしない。代わりに、複数のパラメータ（parameters）によって振る舞いが決定される柔軟なプログラムを定義する。そして、選ばれた性能評価指標に関してプログラムの性能を向上させる、考え得る最良のパラメータ値をデータセットを用いて決定する。

パラメータは、回すことでプログラムの振る舞いを操作できるつまみだと考えることができる。パラメータが固定されると、そのプログラムをモデル（model）と呼ぶ。パラメータを操作するだけで生成できるすべての異なるプログラム（入力から出力へのマッピング）の集合を、モデルのクラス（family）と呼ぶ。そして、データセットを使用してパラメータを選択する「メタプログラム」を学習アルゴリズム（learning algorithm）と呼ぶ。

学習アルゴリズムを導入して進める前に、入出力の正確な性質を突き止め、適切なモデルクラスを選択することで、問題を正確に定義しなければならない。この場合、私たちのモデルは音声断片を入力として受け取り、{yes, no} の中からの選択を出力として生成する。すべてが計画通りに進めば、その断片にウェイクワードが含まれているかどうかについてのモデルの推測は概ね正しいものになるだろう。

適切なモデルクラスを選択すれば、モデルが「Alexa」という単語を聞くたびに「yes」を返すような、つまみの設定が一つ存在するはずである。ウェイクワードの正確な選択は任意であるため、おそらく十分な表現力を持つモデルクラスが必要になる。別のつまみの設定により、「Apricot」という単語を聞いたときにだけ「yes」を返すことができるようにするためだ。「Alexa」の認識と「Apricot」の認識は、直感的に似たタスクに思えるため、同じモデルクラスが

適していると予想される。ただし、根本的に異なる入出力、例えば画像からキャプションへのマッピングや、英語の文から中国語の文へのマッピングなどを扱いたい場合は、全く異なるモデルクラスが必要になるかもしれない。

ご想像の通り、すべてのつまみをランダムに設定しただけでは、モデルが「Alexa」や「Apricot」、その他の英単語を認識する可能性は低い。機械学習において、学習（learning）とは、モデルから望む振る舞いを引き出すための、適切なつまみの設定を発見するプロセスである。言い換えれば、データを使ってモデルを訓練（train）する。Fig. 4.1.2 に示すように、訓練プロセスは通常次のようなものになる。

1. 何も有用なことができない、ランダムに初期化されたモデルから始める。
2. データの一部（例えば、音声断片と対応する {yes, no} ラベル）を取得する。
3. それらのデータ例で評価された性能が向上するように、モデルのつまみを微調整する。
4. モデルが素晴らしい性能になるまで、ステップ 2 と 3 を繰り返す。

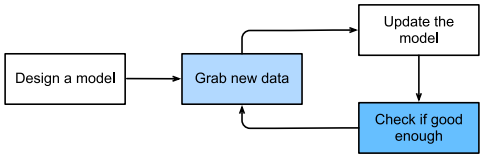


Fig. 4.1.2 典型的な訓練プロセス

要約すると、ウェイクワード認識器をコーディングするのではなく、大規模なラベル付きデータセットを提示されれば、ウェイクワードを認識するように学習できるプログラムをコーディングする。データセットを提示することでプログラムの振る舞いを決定するこの行為は、データを用いたプログラミング（programming with data）と考えることができる。つまり、猫と犬の多数のデータ例を機械学習システムに提供することで、猫検出器を「プログラム」できる。これにより、検出器は最終的に、猫であれば非常に大きな正の数を出力し、犬であれば非常に大きな負の数を出力し、確信が持てない場合はゼロに近い数を出力するように学習する。これは機械学習でできることのほんの一部に過ぎない。後で詳細に説明するディープラーニングは、機械学習の問題を解決するための多くの一般的な手法の中の一つに過ぎない。

4.2 主要なコンポーネント

ウェイクワードの例では、音声断片と 2 値のラベルから構成されるデータセットについて説明し、断片から分類へのマッピングを近似するモデルをどのように訓練するかについて大まかに述べた。未知の指定されたラベルを、ラベルが既知のデータ例から構成されるデータセットが与えられた既知の入力に基づい

て予測しようとするこのような問題は、教師あり学習（supervised learning）と呼ばれる。これは、多くの種類の機械学習の問題の中の一つに過ぎない。他の種類を探求する前に、どのような機械学習の問題に取り組む場合でも常に付きまとういくつかの中核的なコンポーネントに当てはめて、より詳しく説明したい。

1. 学習元となるデータ（data）。
2. データをどのように変換するかというモデル（model）。
3. モデルがどれくらいうまく（あるいは悪く）機能しているかを定量化する目的関数（objective function）。
4. 目的関数を最適化するためにモデルのパラメータを調整するアルゴリズム（algorithm）。

4.2.1 データ

データなしにデータサイエンスを行うことはできないのは言うまでもない。データとは正確には何であるかを考察することで数百ページを費やすこともできるが、ここでは、私たちが扱うデータセットの重要な性質に焦点を当てる。一般的に、私たちはデータ例の集合を扱う。データを有用に利用するためには、通常、適切な数値表現を考案する必要がある。各データ例（example、あるいはデータポイント（data point）、データインスタンス（data instance）、サンプル（sample））は、モデルがそれに基づいて予測を行うために使用する、特徴量（features、時には共変量（covariates）や入力（inputs）とも呼ばれる）と呼ばれる属性の集合から構成される。教師あり学習の問題において、私たちの目標は、モデルの入力には含まれない、ラベル（label、あるいはターゲット（target））と呼ばれる特別な属性の値を予測することである。

もし画像データを扱う場合、各データ例は個々の写真（特徴量）と、その写真が属するカテゴリを示す数値（ラベル）から構成されるかもしれない。写真は、各ピクセル位置における赤、緑、青の光の明るさを示す数値の 3 つのグリッドとして数値的に表現される。例えば、200×200 ピクセルのカラー写真は $200 \times 200 \times 3 = 120000$ 個の数値から構成される。

あるいは、電子カルテデータを扱い、ある患者が今後 30 日生存する可能性を予測するタスクに取り組むかもしれない。ここで特徴量は、容易に入手可能な属性と頻繁に記録される測定値の集合になり、これには年齢、バイタルサイン、併存疾患、現在の投薬、最近の処置が含まれる。訓練に使用できるラベルは、過去のデータにおいて各患者が 30 日以内に生存したかどうかを示す 2 値である。

このような場合、すべてのデータ例が同じ数の数値特徴量によって特徴付けられるとき、入力は固定長のベクトルであると言い、そのベクトルの（一定の）長さをデータの次元数（dimensionality）と呼ぶ。ご想像の通り、固定長の入力は便利であり、悩むべき複雑さを一つ減らしてくれる。しかし、すべてのデータを固定長ベクトルの形で簡単に表現できるわけではない。顕微鏡画像は標

準的な機器から得られることを期待できるが、インターネットからマイニングした画像すべてが同じ解像度や形状を持っているとは期待できない。画像については、標準サイズへの切り抜きを検討するかもしれないが、その戦略には限界がある。切り抜かれた部分の情報を失うリスクがあるからだ。さらに、テキストデータは、固定長表現に対してより頑強に抵抗する。Amazon、IMDb、TripAdvisor などの E コマースサイトに残されたカスタマーレビューを考えてみよう。「ひどい！」という短文のものもあれば、何ページにもわたって長々と書かれているものもある。従来の手法と比較したディープラーニングの大きな利点の一つは、現代のモデルが可変長のデータを比較的優雅に処理できることである。

一般的に、私たちが持つデータが多ければ多いほど、仕事は簡単になる。データが多ければ、より強力なモデルを訓練でき、既存の仮定に頼る割合を減らすことができる。（比較的小さな）データからビッグデータへの時代（regime）の変化は、現代のディープラーニングの成功の大きな要因である。この点を徹底させるために言えば、ディープラーニングにおいて最も魅力的なモデルの多くは、大規模なデータセットなしには機能しない。小規模なデータの時代で機能するモデルもあるかもしれないが、それらは従来の手法よりも優れているわけではない。

最後に、大量のデータがあり、それを巧みに処理するだけでは十分ではない。私たちは正しい（right）データを必要としている。データにエラーが多かったり、選択された特徴量が興味のあるターゲットの数値を予測できなければ、学習は失敗するだろう。この状況は「*garbage in, garbage out*（ごみを入れたら、ごみが出てくる）」という決まり文句でよく言い表される。さらに、予測性能の低さだけが潜在的な結果ではない。予測的取り締まり、履歴書のスクリーニング、融資に使用されるリスクモデルなど、機械学習の機密性の高いアプリケーションでは、ごみデータがもたらす結果に特に警戒しなければならない。よく発生する障害モードの一つは、特定のグループの人々が訓練データに表現されていないデータセットに関するものである。黒い肌を見たことがない皮膚がん認識システムを適用する場合を想像してほしい。データが一部のグループを過小評価しているだけでなく、社会的な偏見を反映している場合にも失敗が発生する。例えば、過去の採用決定を用いて履歴書をスクリーニングするための予測モデルを訓練すると、機械学習モデルは意図せずして歴史的な不平等を捉え、自動化してしまう可能性がある。データサイエンティストが積極的に共謀したり、あるいは認識したりしていなくても、これらすべてが起こり得ることに注意してほしい。

4.2.2 モデル

ほとんどの機械学習は、ある意味でデータを変換すること（transforming）を伴う。写真を取り込み、笑顔度を予測するシステムを構築したいと考えるかもしれない。あるいは、一連のセンサーの読み取り値を取り込み、その読み取り値が正常か異常かを予測したいと考えるかもしれない。私たちが言うモデル（model）とは、ある種類のデータを取り込み、おそらく異なる種類の予測を出力する計算メカニズムを指す。特に、データから推定可能な統計モデル

（statistical models）に関心がある。単純なモデルは、適切な単純な問題に完全に対処できる能力を持っているが、私たちが本書で焦点を当てる問題は、古典的な手法の限界を広げるものである。ディープラーニングが古典的なアプローチと区別される主な理由は、それが焦点を当てる強力なモデルの集合にある。これらのモデルは、データを連続的に（successive）何度も変換し、それらを上から下へと連鎖（chain）させたものであり、それがディープラーニング（Deep learning, 深層学習）の名前の由来となっている。深いモデルについて議論する過程で、より伝統的な手法についても議論する。

4.2.3 目的関数

前述のように、機械学習を経験からの学習として紹介した。ここにおける学習とは、あるタスクにおいて時間をかけて改善していくことを意味する。しかし、何が改善を構成するかは誰が言うのだろうか？モデルを更新する提案を我々が行い、その提案が改善を構成するかどうかについて反対する人がいるかもしれないと想像できるだろう。

学習マシンの正式な数学的システムを開発するためには、私たちのモデルがどれだけ優れているか（あるいは悪いか）についての正式な尺度を持つ必要がある。機械学習において、さらに一般的には最適化において、これらを目的関数（objective functions）と呼ぶ。慣習により、目的関数は通常、低い値であるほど良いものとして定義される。これは単なる慣習に過ぎない。高い値ほど良い関数であっても、符号を反転させることで、定性的には完全に同一であるが低い値ほど良い新しい関数に変換できる。低い値が良いとして私たちが選択するため、これらの関数は損失関数（loss functions）と呼ばれることがある。

数値を予測しようとする場合、最も一般的な損失関数は二乗誤差（squared error）である。すなわち、予測と正解ターゲット（ground truth target）の差の二乗である。分類において最も一般的な目的は、誤り率（error rate）、つまり私たちの予測が正解と一致しないデータ例の割合を最小化することである。一部の目的（例えば、二乗誤差）は簡単に最適化できるが、他の目的（例えば、誤り率）は、微分不可能であるか、または他の複雑な理由から直接最適化することが困難である。このような場合、代わりに代替目的（surrogate objective）を最適化することが一般的である。

最適化の最中、私たちは損失をモデルのパラメータの関数と考え、訓練データセットを定数として扱う。訓練のために収集されたいくつかのデータ例からなる集合上で生じる損失を最小化することで、私たちのモデルのパラメータの最適な値を学習する。しかし、訓練データでうまく機能しても、未知のデータでうまく機能するとは限らない。そのため、利用可能なデータを通常 2 つのパーティションに分割したいと考えるだろう。すなわち、モデルのパラメータを学習するための訓練データセット（training dataset または 訓練セット training set）と、評価のために確保されるテストデータセット（test dataset または テストセット test set）である。最終的に、私たちは通常、両方のパーティションにおけるモデルのパフォーマンスを報告する。訓練のパフォーマンスは、学生が実際の最終試験の準備に使用する模擬試験で達成した点数に例えることができる。結果が励みになるものであっても、それが最終試験での成功を保証するものでは

ない。学習の過程で、学生は模擬問題の暗記を始め、そのトピックを習得しているように見えても、実際の最終試験で未知の問題に直面するとつまずくかもしれない。モデルが訓練セットではうまく機能するが、未知のデータに一般化できない場合、訓練データに過学習（overfitting）していると言う。

4.2.4 最適化アルゴリズム

データソースと表現、モデル、そして明確に定義された目的関数が揃えば、損失関数を最小化するための可能な限り最良のパラメータを探索できるアルゴリズムが必要になる。ディープラーニングで一般的な最適化アルゴリズムは、勾配降下法（gradient descent）と呼ばれるアプローチに基づいている。手短に言えば、各ステップにおいて、この手法は各パラメータについて、そのパラメータをほんのわずかだけ摂動（perturb）させた場合に訓練セットの損失がどのように変化するかを確認する。そして、損失を下げる方向にパラメータを更新する。

4.3 機械学習の問題の種類

動機付けの例で挙げたウェイクワードの問題は、機械学習が対処できる多くの問題の中の一つに過ぎない。読者の関心をさらに高め、本書を通じて使われる共通の言語を提供するために、ここで機械学習の問題の全体像について幅広く概観する。

4.3.1 教師あり学習

教師あり学習（Supervised learning）は、特徴量とラベルの両方を含むデータセットが与えられ、入力特徴量が与えられたときにラベルを予測するモデルを作成することを求められるタスクを指す。各「特徴量-ラベル」のペアはデータ例（example）と呼ばれる。文脈が明らかな場合、対応するラベルが未知であっても、入力の集合を指してデータ例という用語を使用することがある。パラメータを選択するために、私たち（教師 / supervisors）がラベル付きデータ例からなるデータセットをモデルに提供するため、教師（supervision）の概念が登場する。確率論的な用語で言えば、私たちは通常、入力特徴量が与えられたときのラベルの条件付き確率を推定することに関心がある。これはいくつかのパラダイムの中の一つに過ぎないが、教師あり学習は、産業界における機械学習の成功事例の大部分を占めている。その理由の一つは、多くの重要なタスクが、特定の利用可能なデータの集合が与えられたときに、未知の何かの確率を推定することとして明確に記述できるからである。：

- CT（コンピュータ断層撮影）画像が与えられ、がんか否かを予測する。
- 英語の文が与えられ、フランス語の正しい翻訳を予測する。
- 今月の財務報告データに基づき、来月の株価を予測する。

すべての教師あり学習の問題は「入力特徴量が与えられたときにラベルを予測する」という単純な説明で捉えられるが、教師あり学習自体は多様な形態をとり得り、（他の考慮事項に加えて）入出力の種類、サイズ、および量に応じて大量のモデル化の決定を必要とする。例えば、任意の長さのシーケンスを処理する場合と固定長のベクトル表現を処理する場合では異なるモデルを使用する。本書を通じて、これらの問題の多くについて深く掘り下げていく。

非公式には、学習プロセスは次のようなものである。まず、特徴量が既知の巨大なデータ例の集合を用意し、そこからランダムな部分集合を選択して、それぞれについて正解ラベルを取得する。これらのラベルは、すでに収集された利用可能なデータ（例えば、患者が翌年以内に死亡したか？など）であることもあれば、データをラベル付けするために人間のアノテーターを雇用する必要がある場合（例えば、画像をカテゴリに割り当てるなど）もある。これらの入力と対応するラベルが一緒になって、訓練セットを構成する。訓練データセットを教師あり学習アルゴリズムに入力する。このアルゴリズムは、データセットを入力として受け取り、別の関数、すなわち学習済みモデルを出力する関数である。最後に、これまで見たことのない入力を学習済みモデルに与え、その出力を対応するラベルの予測として使用できる。全体のプロセスを Fig. 4.3.1 に示す。

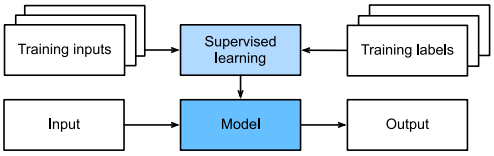


Fig. 4.3.1 教師あり学習

回帰

おそらく、最も理解しやすい教師あり学習のタスクは回帰（regression）である。例えば、住宅販売のデータベースから収集したデータの集合を考えてみよう。各行が異なる住宅に対応し、各列が住宅の面積（平方フィート）、寝室の数、バスルームの数、町の中心部までの徒歩分数などの関連する属性に対応する表を構築するかもしれない。このデータセットにおいて、各データ例は特定の住宅であり、対応する特徴量ベクトルは表の中の 1 行となる。もしあなたがニューヨークやサンフランシスコに住んでいて、Amazon、Google、Microsoft、または Facebook の CEO ではない場合、あなたの家の（面積、寝室数、バスルーム数、徒歩分数）という特徴量ベクトルは $[600, 1, 1, 60]$ のようになるかもしれない。しかし、もしピッツバーグに住んでいるなら、それは $[3000, 4, 3, 10]$ のようになるかもしれない。このような固定長の特徴量ベクトルは、ほとんどの古典的な機械学習アルゴリズムにとって不可欠である。

ある問題を回帰にするものは、実際にはターゲットの形式である。あなたが新しい家を探しているとしよう。上記のような特徴量が与えられたときに、家の公正な市場価値を推定したいと考えるかもしれない。ここでのデータは過去の住宅物件情報から構成され、ラベルは観測された販売価格になるかもしれな

い。ラベルが（ある区間内であっても）任意の数値をとるとき、私たちはこれを回帰（regression）問題と呼ぶ。目標は、実際のラベル値に密接に近似する予測を出力するモデルを構築することである。

多くの実用的な問題は、回帰問題として簡単に記述できる。ユーザーが映画につける評価（レーティング）を予測することは回帰問題と考えられ、もし2009年にこの偉業を達成する素晴らしいアルゴリズムを設計していれば、100万ドルのNetflix賞金[13]を獲得していたかもしれない。患者の入院期間の予測も回帰問題である。経験則として、「どれくらい（*how much?*）」や「いくつ（*how many?*）」を問う問題は、おそらく回帰である。例えば：* この手術は何時間かかるだろうか？* 今後6時間でこの町にはどれくらいの雨が降るだろうか？

[13] 

たとえこれまで機械学習に取り組んだことがなくても、おそらく非公式に回帰問題に取り組んだことがあるだろう。例えば、ご自宅の排水管を修理してもらい、業者が下水管から泥を取り除くのに3時間かかったと想像してほしい。その後、350ドルの請求書が送られてきた。ここで、あなたの友人が同じ業者を2時間雇い、250ドルの請求書を受け取ったと想像してほしい。その後、知り合いから今後の泥除去の請求額はどれくらいになるかと聞かれたら、例えば、労働時間が長いほど料金が高くなるなどの合理的な仮定を立てるかもしれない。また、基本料金があり、業者がそこから時間単位で請求していると仮定するかもしれない。これらの仮定が当てはまるのであれば、これら2つのデータ例が与えられれば、業者の料金体系、すなわち1時間あたり100ドルに、家を訪問するための基本料金50ドルを加えた料金体系を特定できる。ここまで理解できたのなら、線形回帰の背後にある大まかなアイデアをすでに理解しているということである。

この場合、業者の価格と完全に一致するパラメータを算出できた。場合によっては、例えば、変動の一部が2つの特徴量以外の要因から生じている場合など、これが不可能なこともある。このような場合、予測値と観測値の間の距離を最小化するモデルを学習しようとする。本書の大部分の章では、二乗誤差損失関数の最小化に焦点を当てる。後でわかるように、この損失はデータがガウスノイズによって破損されているという仮定に対応している。

分類

回帰モデルは「いくつ（*how many?*）」の質問に対処するのに優れているが、多くの問題はこのテンプレートにうまく収まらない。例えば、モバイルアプリの小切手スキャン機能を開発したいと考えている銀行を考えてみよう。理想的には、顧客が小切手の写真を撮影するだけで、アプリが画像からテキストを自動的に認識するだろう。それぞれの手書き文字に対応する画像のパッチを切り出すある程度の能力があるとした場合、残された主要なタスクは、既知の集合の中で、それぞれの画像パッチにどの文字が描かれているかを決定することになる。このような「どれ（*which one?*）」を問う問題は分類（classification）と呼ばれ、多くの技術が引き継がれるものの、回帰で使用されるものとは異なるツール群を必要とする。

分類では、モデルが特徴量（例えば画像のピクセル値）を見て、ある離散的な選択肢の集合の中で、データ例がどのカテゴリ（category、あるいはクラス class とも呼ばれる）に属するかを予測することを求める。手書き数字の場合、0 から 9 までの数字に対応する 10 個のクラスがあるかもしれない。分類の最も単純な形態は、2 つのクラスしかない場合であり、この問題を私たちは *2* 値分類（binary classification）と呼ぶ。例えば、データセットが動物の画像で構成され、ラベルが {cat, dog} というクラスである場合などだ。回帰では数値をを出力する回帰器（regressor）を求めたが、分類では出力を予測されるクラスへの割り当てとする分類器（classifier）を求める。

本書が技術的になるにつれて説明する理由から、「猫」か「犬」かといった確固たる（firm）カテゴリへの割り当てしか出力できないモデルを最適化することは困難な場合がある。このような場合、モデルを確率の言語で表現する方がはるかに簡単であることが多い。データ例の特徴量が与えられたとき、私たちのモデルはそれぞれの可能なクラスに確率を割り当てる。クラスが {cat, dog} である動物分類の例に戻ると、分類器は画像を見て、その画像が猫である確率を 0.9 と出力するかもしれない。この数字は、分類器がその画像には猫が描かれていると 90% の確信を持っていると解釈できる。予測されたクラスの確率の大きさは、不確実性の概念を伝える。利用できる概念はこれだけではなく、より高度なトピックを扱う章で他の概念についても議論する。

可能なクラスが 3 つ以上ある場合、その問題を多クラス分類（multiclass classification）と呼ぶ。一般的な例としては、手書き文字認識 {0, 1, 2, ... 9, a, b, c, ...} が挙げられる。回帰問題には二乗誤差損失関数を最小化することで取り組んだが、

分類問題における一般的な損失関数は交差エントロピー（cross-entropy）と呼ばれ、後の章で情報理論を紹介する際にその名前の謎が解き明かされるだろう。

最も可能性の高いクラスが、必ずしも意思決定に使用するクラスとは限らないことに注意してほしい。Fig. 4.3.2 に示すように、裏庭で美しいキノコを見つけたと仮定しよう。

さて、分類器を構築し、写真に基づいてキノコに毒があるかどうかを予測するように訓練したと仮定する。この毒検出分類器が、Fig. 4.3.2 に死の帽子（タマゴテングタケ）が写っている確率を 0.2 と出力したとしよう。言い換えれば、分類器はこのキノコがタマゴテングタケではないと 80% の確信を持っているということだ。それでも、それを食べるのは愚か者だけだろう。なぜなら、美味しい夕食という確実な利益は、20% の確率で死に至るリスクに見合うものではないからだ。換言すれば、不確実なリスクの影響が利益をはるかに上回っている。したがって、このキノコを食べるかどうかを決定するためには、想定される結果とそれぞれに関連する損益の両方に依存する、各行動に関連する予測される損害を計算する必要がある。この場合、キノコを食べることで生じる損害は $0.2 \times \infty + 0.8 \times 0 = \infty$ になるかもしれないが、キノコを捨てることの損失は $0.2 \times 0 + 0.8 \times 1 = 0.8$ である。私たちの警戒は正当化された。菌類学者なら誰でも言うように、Fig. 4.3.2 のキノコは実際にはタマゴテングタケである。



Fig. 4.3.2 タマゴテングタケ—食べてはいけない！

分類は、2 値分類や多クラス分類よりもはるかに複雑になることがある。例えば、階層的に構造化されたクラスに対処する分類のバリエーションがいくつかある。そのような場合、すべてのエラーが等しいわけではない。もし間違えなければならないのであれば、遠いクラスよりも関連するクラスに誤分類することを好むかもしれない。通常、これは階層的分類（hierarchical classification）と呼ばれる。ヒントとして、動物相を階層構造に整理したリンネ[14] を思い浮かべるとよいだろう。

14

動物の分類の場合、プードルをシュナウザーと間違えるのはそれほど悪くないかもしれないが、プードルを恐竜と混同した場合、私たちのモデルは莫大なペナルティを支払うことになるだろう。どの階層が関連するかは、モデルをどのように使用する予定かによって異なる場合がある。例えば、ガラガラヘビとガータースネークは系統樹上で近いかもしれないが、ガラガラヘビをガータースネークと間違えることは致命的な結果を招く可能性がある。

タグ付け

一部の分類問題は、2 値分類や多クラス分類の設定にきれいに収まる。例えば、一般的な 2 値分類器を訓練して、猫と犬を区別することができる。コンピュータビジョンの現状を考慮すれば、既製のツールを使ってこれを簡単に行うことができる。それにもかかわらず、私たちのモデルがどれほど正確になっても、分類器が、4 匹の動物が登場する有名なドイツの童話ブレーメンの音楽隊の画像（Fig. 4.3.3）に遭遇したときにトラブルに陥るかもしれない。

ご覧の通り、この写真には猫、雄鶏、犬、ロバが映っており、背景にはいくつかの木がある。もしこのような画像に遭遇することが予想されるのであれば、多クラス分類は適切な問題の定式化ではないかもしれない。代わりに、画像には猫、犬、ロバ、そして雄鶏が描かれていると言う選択肢をモデルに与えたいと考えるかもしれない。

互いに排他的ではないクラスを予測するように学習する問題は、マルチラベ



Fig. 4.3.3 ロバ、犬、猫、そして雄鶏

ル分類（multi-label classification）と呼ばれる。自動タグ付けの問題は通常、マルチラベル分類の観点から説明するのが最も適切である。テクノロジーブログの投稿に人々が適用するかもしれないタグ、例えば「機械学習」、「テクノロジー」、「ガジェット」、「プログラミング言語」、「Linux」、「クラウドコンピューティング」、「AWS」について考えてみよう。一般的な記事には 5〜10 個のタグが適用されるかもしれない。通常、タグはいくらかの相関構造を示す。「クラウドコンピューティング」に関する投稿は「AWS」に言及する可能性が高く、「機械学習」に関する投稿は「GPU」に言及する可能性が高い。

時には、このようなタグ付け問題は膨大なラベル集合に依存することがある。国立医学図書館（The National Library of Medicine）は、PubMed でインデックス付けされる各記事を、Medical Subject Headings（MeSH）オントロジー（約 28,000 のタグの集合）から抽出されたタグの集合と関連付ける多くの専門アノテーターを雇用している。記事に正しくタグ付けすることは、研究者が文献の網羅的なレビューを行うことを可能にするため重要である。これは時間がかかるプロセスであり、通常、アーカイブ化とタグ付けの間には 1 年間の遅れがある。機械学習は、各記事が適切な手動レビューを受けるまでの間、暫定的なタグを提

供できる。実際、BioASQ という組織は数年間にわたり、このタスクのためのコンペティションを開催[15] してきた。

15 

検索

情報検索の分野では、アイテムの集合に対してランク（順位）を課すことがよくある。ウェブ検索を例にとってみよう。目標は、特定のページが検索クエリに関連しているかどうかを判断することではなく、むしろ、関連する結果の集合の中で、どれを特定のユーザーに最も目立つように表示するかを決定することである。これを行う 1 つの方法は、まず集合内のすべての要素にスコアを割り当て、次に最高評価の要素を取得することである。Google 検索エンジンの背後にあるオリジナルの秘伝のソースであるPageRank[16] は、このようなスコアリングシステムの初期の例であった。奇妙なことに、PageRank が提供するスコアリングは実際のクエリには依存していなかった。代わりに、単純な関連性フィルターに依存して関連する候補の集合を特定し、その後 PageRank を使用してより権威のあるページを優先順位付けしていた。今日では、検索エンジンは機械学習や行動モデルを使用して、クエリに依存する関連性スコアを取得している。このテーマに特化した学術会議さえ存在する。

16 

推薦システム

推薦システム（Recommender systems）は、検索やランキングに関連するもう一つの問題設定である。ユーザーに関連するアイテムの集合を表示するという目標という点では、これらの問題は似ている。主な違いは、推薦システムの文脈においては特定のユーザーへのパーソナライズ（personalization）が強調されることである。例えば、映画の推薦において、SF ファンのための結果ページと、ピーター・セラーズのコメディ愛好家のための結果ページは大きく異なるかもしれない。小売商品、音楽、ニュースの推薦など、他の推薦設定でも同様の問題が発生する。

顧客が明示的なフィードバックを提供し、彼らが特定の製品をどれくらい気に入ったかを伝えることがある（例えば、Amazon、IMDb、Goodreads での製品評価やレビューなど）。一方で、プレイリストのタイトルをスキップするなど暗黙のフィードバックを提供することもあり、これは不満を示しているか、あるいは単にその曲が状況に不適切であったことを示しているだけかもしれない。最も単純な定式化では、これらのシステムは、期待される星評価や、特定のユーザーがあるアイテムを購入する確率といった、何らかのスコアを推定するように訓練される。

そのようなモデルが与えられれば、任意のユーザーに対して、最大のスコアを持つオブジェクトの集合を取得でき、それをそのユーザーに推薦できる。本番環境（プロダクション）のシステムははるかに高度であり、そのようなスコアを計算する際に、ユーザーの詳細なアクティビティやアイテムの特性を考慮に入れている。Fig. 4.3.4 は、アストンの好みを捉えるように調整されたパーソナライズアルゴリズムに基づいて、Amazon から推薦されたディープラーニング専門書を表示している。

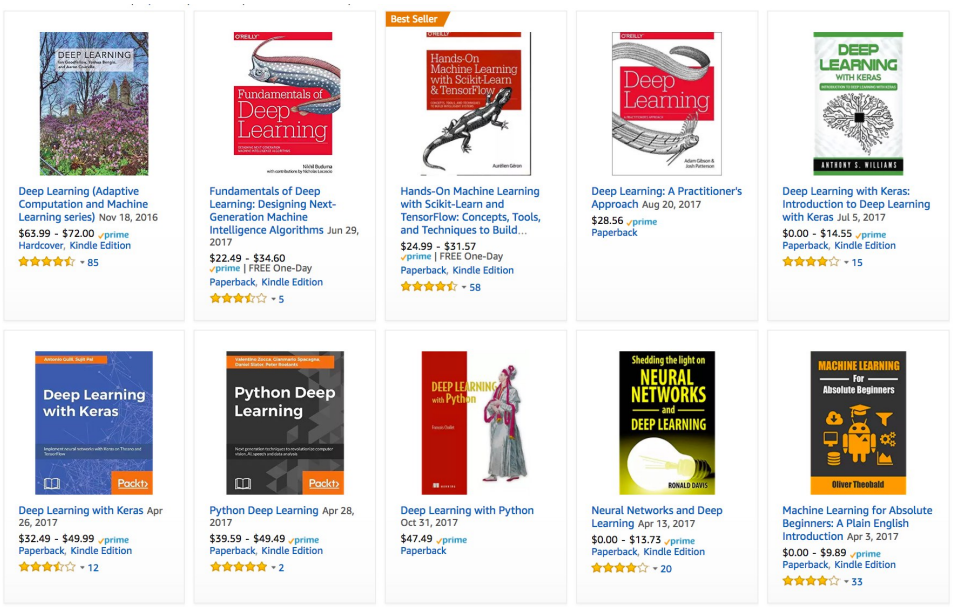


Fig. 4.3.4 Amazon が推奨するディープラーニング書籍

予測モデルの上に素朴に構築された推薦システムは、その莫大な経済的価値にもかかわらず、深刻な概念上の欠陥を抱えている。まず第一に、私たちは打ち切りフィードバック（censored feedback）しか観測できない。ユーザーは、強い感情を抱いた映画を優先的に評価する。例えば、5 段階評価において、アイテムが 1 つ星や 5 つ星の評価を多く受ける一方で、3 つ星の評価が著しく少ないことに気付くかもしれない。さらに、現在の購買習慣は、現在導入されている推薦アルゴリズムの結果であることが多いが、学習アルゴリズムは必ずしもこの詳細を考慮に入れているわけではない。したがって、推薦システムが特定のアイテムを優先的にプッシュし、それが（購入数の多さから）より良いものと見なされ、その結果さらに頻繁に推薦されるというフィードバックループが形成される可能性がある。これら多くの問題、すなわち打ち切り、インセンティブ、およびフィードバックループにどのように対処するかは、重要な未解決の研究課題である。

シーケンス学習

ここまで、一定数の入力があり、一定数の出力を生成する問題を見てきた。例えば、面積、寝室数、バスルーム数、繁華街までの移動時間という固定された特徴量の集合から住宅価格を予測することを検討した。また、固定次元の画像から、それが一定数のクラスのそれぞれに属する予測確率へのマッピングや、ユーザー ID と製品 ID のみに基づいて購入に関連する星評価を予測することについても議論した。これらの場合、モデルが訓練された後、各テスト例がモデルに入力されると、それはすぐに忘れられる。連続する観測は独立していると仮定し、したがってこの文脈を保持する必要はないと考えた。

しかし、ビデオの断片にはどのように対処すべきだろうか？この場合、各断片は異なる数のフレームから構成されるかもしれない。そして、前後のフレームを考慮すれば、各フレームで何が起こっているかの予測はもっと強力になるかもしれない。言語についても同じことが言える。例えば、ディープラーニングで人気のある問題の一つに機械翻訳がある。あるソース言語の文を取り込み、別の言語での翻訳結果を予測するタスクである。

このような問題は医療においても発生する。集中治療室の患者を監視し、今後24 時間以内に死亡するリスクがある閾値を超えた場合に警告を発するモデルが欲しいと考えるかもしれない。ここでは、患者の履歴について知っていることすべてを 1 時間ごとに捨ててしまうことはしない。なぜなら最新の測定値のみに基づいて予測を立てたくないからだ。

このような疑問は、機械学習の最も魅力的な応用分野の 1 つであり、これらはシーケンス学習（sequence learning）の事例である。これらのモデルは、入力のシーケンスを取り込むか、出力のシーケンスを生成する（あるいはその両方）のいずれかを行う必要がある。具体的には、シーケンス・ツー・シーケンス学習（sequence-to-sequence learning）は、入出力の両方が可変長のシーケンスから構成される問題を考慮する。例としては、機械翻訳や音声からテキストへの書き起こしなどがある。すべての種類のシーケンス変換を考慮することは不可能だが、以下の特別なケースは言及する価値がある。

タグ付けとパース。これは、テキストシーケンスに属性で注釈を付けることを含む。ここでは、入出力は整列（aligned）している。つまり、数が同じであり、対応する順序で発生する。例えば、品詞（*PoS*）タグ付けでは、文中のすべての単語に対応する品詞、すなわち「名詞」や「直接目的語」などを注釈する。あるいは、隣接するどの単語のグループが人、場所、または組織のような固有表現を指しているかを知りたいかもしれない。下の漫画のように単純な例では、文中のいずれかの単語が固有表現（「Ent」としてタグ付け）の一部であるかどうかだけを示したいかもしれない。

```
Tom has dinner in Washington with Sally
Ent  -    -    -     Ent      -    Ent
```

自動音声認識。音声認識では、入力シーケンスは話者の録音音声（Fig. 4.3.5）であり、出力は話者が言ったことの書き起こしである。課題は、テキストよりもはるかに多くのオーディオフレーム（通常、音声は 8kHz または 16kHz でサンプリングされる）が存在することである。つまり、数千のサンプルが単一の発話された単語に対応する可能性があるため、音声とテキストの間に 1：1 の対応関係はない。これらは出力が入力よりもはるかに短いシーケンス・ツー・シーケンス学習の問題である。

人間は低品質の音声からでも音声を認識することに驚くほど優れているが、コンピュータに同じ離れ業を実行させることは手ごわい課題である。

音声合成（Text to Speech）。これは自動音声認識の逆である。ここでは、入力が

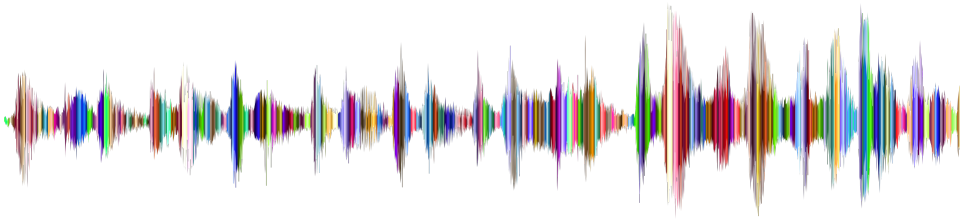


Fig. 4.3.5 音声録音における-D-e-e-p- L-ea-r-ni-ng-

テキストであり、出力がオーディオファイルである。この場合、出力は入力よりもはるかに長くなる。

機械翻訳。対応する入出力が同じ順序で発生する音声認識の場合とは異なり、機械翻訳では、整列されていない（unaligned）データが新たな課題をもたらす。ここでは、入出力のシーケンスの長さが異なる可能性があり、それぞれのシーケンスの対応する領域が異なる順序で現れる可能性がある。動詞を文末に置くというドイツ語の独特な傾向を示す、以下のわかりやすい例を考えてみよう。

ドイツ語:	Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
英語:	Have you already looked at this excellent textbook?
誤った対応付け:	Have you yourself already this excellent textbook looked at?

他の学習タスクでも多くの関連する問題が発生する。例えば、ユーザーがウェブページを読む順序を決定することは、2 次元のレイアウト解析問題である。対話問題はあらゆる種類のさらなる複雑さを示し、次に何を言うべきかを決定するには、現実世界の知識と、長い時間的距離にわたる会話の過去の状態を考慮する必要がある。そのようなトピックは活発な研究分野である。

4.3.2 教師なし学習と自己教師あり学習

これまでの例は教師あり学習に焦点を当てており、ここでは特徴量と対応するラベル値の両方を含む巨大なデータセットをモデルに与えた。教師あり学習器は、極めて専門化された仕事と、極めて独裁的な上司を持っていると考えることができる。上司は学習器の肩越しに立ち、彼らが状況から行動へのマッピングを学習するまで、あらゆる状況で正確に何をすべきかを彼らに指示する。そのような上司の下で働くのはかなり退屈に聞こえる。一方で、そのような上司を満足させるのは非常に簡単である。可能な限り早くパターンを認識し、上司の行動を模倣すればよいだけだ。

反対の状況を考えてみると、あなたに何をしてほしいのか全くわかっていない上司の下で働くのはイライラするかもしれない。しかし、もしあなたがデータサイエンティストを目指しているなら、それに慣れた方がよい。上司は巨大なデータのダンプを渡して、「それでデータサイエンスをしてくれ！」とだけ言うかもしれない。曖昧に聞こえるが、それは実際に曖昧だからだ。このクラスの問題を私たちは教師なし学習（unsupervised learning）と呼び、私たちが尋ね

ることができる質問の種類と数は、私たちの創造性によってのみ制限される。後の章で教師なし学習の手法について取り上げる。ここでは食欲をそそるために、あなたが尋ねるかもしれない次のようないくつかの質問について説明する。

- データを正確に要約する少数のプロトタイプを見つけることができるだろうか？写真の集合が与えられたとき、それらを風景写真、犬、赤ちゃん、猫、山頂の写真にグループ化できるだろうか？同様に、ユーザーのブラウジングアクティビティの集合が与えられたとき、それらを類似した行動を持つユーザーにグループ化できるだろうか？この問題は一般的にクラスタリング（clustering）として知られている。

- データの関連する特性を正確に捉える少数のパラメータを見つけることができるだろうか？ボールの軌道は、ボールの速度、直径、質量によってうまく表現される。仕立屋は、服を合わせる目的で、人間の体型をかなり正確に表現する少数のパラメータを開発した。これらの問題は部分空間推定（subspace estimation）と呼ばれる。依存関係が線形である場合、それは主成分分析（principal component analysis）と呼ばれる。

- 象徴的な特性がうまく適合するように、ユークリッド空間における（任意に構造化された）オブジェクトの表現は存在するだろうか？これは、「ローマ」−「イタリア」+「フランス」=「パリ」のように、エンティティとその関係を記述するために使用できる。

- 私たちが観察するデータの多くについて、根本的な原因の記述は存在するだろうか？例えば、住宅価格、汚染、犯罪、立地、教育、給与に関する人口統計データがある場合、単に経験的データに基づいてそれらがどのように関連しているかを発見できるだろうか？因果関係（causality）や確率的グラフィカルモデル（probabilistic graphical models）に関わる分野は、そのような問題に取り組んでいる。

- 教師なし学習におけるもう一つの重要で刺激的な最近の進展は、深層生成モデル（deep generative models）の出現である。これらのモデルは、データの密度を明示的または暗黙的に推定する。訓練が終われば、生成モデルを使用して、データ例がどれくらい起こり得るかに従ってスコアリングしたり、学習された分布から合成データをサンプリング（生成）したりできる。生成モデリングにおける初期のディープラーニングのブレークスルーは、変分オートエンコーダ（variational autoencoders）:cite:*Kingma.Welling.2014,rezende2014stochastic* の発明によってもたらされ、敵対的生成ネットワーク（generative adversarial networks, GAN）:cite:*Goodfellow.Pouget-Abadie.Mirza.ea.2014* の開発へと続いた。さらに最近の進歩には、正規化フロー（normalizing flows）:cite:*dinh2014nice,dinh2017density* や拡散モデル（diffusion models）:cite:*sohl2015deep,song2019generative,ho2020denoising,song2021score* などがある。

教師なし学習におけるさらなる進展は、ラベルなしデータの何らかの側面を利用して教師（supervision）を提供する技術である自己教師あり学習（self-supervised learning）の台頭である。テキストの場合、ラベル付けの労力を一切か

けずに、大規模コーパスにおいて周囲の単語（文脈）を使用して、ランダムにマスクされた単語を予測することにより、「空白を埋める」ようにモデルを訓練できる (Devlin *et al.*, 2018)! 画像の場合、同じ画像の 2 つの切り抜かれた領域の間の相対位置を見分けるようにモデルを訓練したり (Doersch *et al.*, 2015)、画像の残りの部分に基づいて隠された部分を予測したり、2 つのデータ例が同じ元の画像の摂動されたバージョンであるかどうかを予測したりできる。自己教師ありモデルはしばしば表現を学習し、その結果得られたモデルを興味のある下流のタスクでファインチューニング（微調整）することによって、その表現がその後活用される。

4.3.3 環境との相互作用

ここまで、データが実際にどこから来るのか、あるいは機械学習モデルが出力を生成したときに実際に何が起こるのかについては議論してこなかった。それは、教師あり学習と教師なし学習がこれらの問題にあまり洗練された方法で対処していないためである。どちらの場合も、最初に大量のデータを確保し、その後環境と再びやり取りすることなくパターン認識マシンの動作を開始する。すべての学習がアルゴリズムが環境から切り離された後に行われるため、これはオフライン学習（offline learning）と呼ばれることがある。例えば、教師あり学習は Fig. 4.3.6 に描かれているような単純な相互作用のパターンを前提としている。

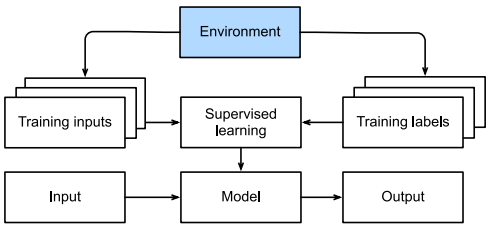


Fig. 4.3.6 教師あり学習のための環境からのデータ収集

このオフライン学習の単純さには魅力がある。利点は、動的な環境との相互作用から生じる複雑さを気にすることなく、世界から切り離された状態でパターン認識について悩むことができることである。しかし、この問題の定式化には限界がある。もしあなたがアシモフのロボット小説を読んで育ったなら、単に予測を行うだけでなく、世界で行動を起こすことができる人工的なインテリジェント・エージェントを想像するだろう。私たちは単なる予測モデルではなく、インテリジェントなエージェント（agents）について考えたい。これは、予測を行うだけでなく、行動（actions）の選択について考える必要があることを意味する。単なる予測とは対照的に、行動は実際に環境に影響を与える。もし私たちがインテリジェント・エージェントを訓練したいのであれば、その行動がエージェントの将来の観測にどのような影響を与えるかを考慮しなければならず、したがってオフライン学習は不適切である。

環境との相互作用を考慮すると、全く新しい一連のモデリングの疑問が開かれる。以下はほんの一例である。* 環境は私たちが過去にしたことを覚えている

か？* 環境は私たちを助けたいと思っているか？（例：音声認識器にテキストを読み上げるユーザー）* 環境は私たちを打ち負かしたいと思っているか？（例：スパムフィルターを回避するためにメールを適応させるスパマー）* 環境には変化するダイナミクスがあるか？例えば、将来のデータは常に過去に似ているのか、それともパターンが時間の経過とともに、自然にあるいは私たちの自動化ツールに反応して変化するのか？

これらの疑問は、訓練データとテストデータが異なる分布シフト（distribution shift）という問題を提起する。私たちの多くが遭遇したことがあるであろうその一例は、宿題がティーチングアシスタントによって作成された一方で、講師が作成した試験を受ける場合である。次に、エージェントが環境と相互作用する学習問題を提示するための強力なフレームワークである、強化学習について簡単に説明する。

4.3.4 強化学習

もし環境と相互作用して行動を起こすエージェントを開発するために機械学習を使用することに興味があるなら、おそらく最終的には強化学習（reinforcement learning）に焦点を当てることになるだろう。これにはロボット工学、対話システム、さらにはビデオゲーム用の人工知能（AI）の開発への応用が含まれる可能性がある。強化学習問題にディープラーニングを適用する深層強化学習（Deep reinforcement learning）の人気は急上昇している。視覚入力のみを使用して Atari のゲームで人間に勝利した画期的な Deep Q-Network (Mnih *et al.*, 2015) や、ボードゲームの囲碁で世界チャンピオンの座を奪った AlphaGo プログラム (Silver *et al.*, 2016) は、2 つの著名な例である。

強化学習は、エージェントが一連のタイムステップにわたって環境と相互作用する問題の非常に一般的な記述を与える。各タイムステップで、エージェントは環境から何らかの観測（observation）を受け取り、その後に何らかのメカニズム（時にはアクチュエータ（actuator）と呼ばれる）を介して環境に送り返される行動（action）を選択しなければならない。そして各ループの後、エージェントは環境から報酬（reward）を受け取る。このプロセスは Fig. 4.3.7 に示されている。エージェントはその後、次の観測を受け取り、次の行動を選択する、というように続く。強化学習エージェントの振る舞いは方策（policy）によって支配される。手短に言えば、方策は環境の観測から行動へのマッピングを行う単なる関数である。強化学習の目標は、優れた方策を導き出すことである。

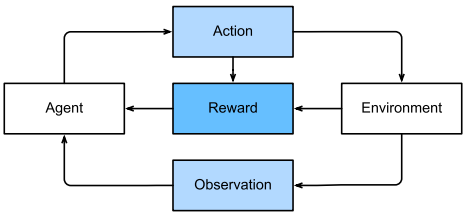


Fig. 4.3.7 強化学習と環境の相互作用

強化学習フレームワークの汎用性はいくら強調してもしすぎることはない。例

えば、教師あり学習は強化学習として書き直すことができる。分類問題があったとしよう。各クラスに対応する 1 つの行動を持つ強化学習エージェントを作成できる。そして、元の教師あり学習問題の損失関数と完全に等しい報酬を与える環境を作成できる。

さらに、強化学習は教師あり学習では対処できない多くの問題にも対処できる。例えば、教師あり学習では、訓練入力には常に正解ラベルが関連付けられていると期待する。しかし、強化学習では、各観測について環境が最適な行動を教えてくれるとは仮定しない。一般に、何らかの報酬が得られるだけである。さらに、環境はどの行動が報酬につながったかさえ教えてくれないかもしれない。

チェスのゲームを考えてみよう。唯一の実際の報酬信号はゲームの最後にやってくる。勝負に勝って例えば 1 の報酬を得るか、負けて例えば –1 の報酬を受け取るときである。そのため、強化学習器は信用割り当て（credit assignment）の問題、すなわち結果に対してどの行動を評価し、または非難すべきかを決定する問題に対処しなければならない。10 月 11 日に昇進した従業員についても同じことが言える。その昇進はおそらく、前年の間に適切に選択された多数の行動を反映している。将来昇進するためには、これまでのどの行動が昇進につながったのかを解明する必要がある。

強化学習器は、部分観測性（partial observability）の問題にも対処しなければならない場合がある。つまり、現在の観測が、あなたの現在の状態のすべてを教えてくれるとは限らない。掃除ロボットが、家の中にある同じ形をした多くのクローゼットの 1 つに閉じ込められたことに気づいたとしよう。ロボットを救出するにはその正確な位置を推測する必要があり、クローゼットに入る前の過去の観測を考慮する必要があるかもしれない。

最後に、どの時点においても、強化学習器は 1 つの優れた方策を知っているかもしれないが、エージェントが試したことのないより優れた方策が他にも多数存在する可能性がある。強化学習器は、知識と引き換えに潜在的にある程度の短期的な報酬を犠牲にしてでも、最良の（現在のところ）既知の戦略を方策として利用（exploit）するか、戦略の空間を探索（explore）するかを常に選択しなければならない。

一般的な強化学習問題は、非常に一般的な設定を持つ。行動はその後の観測に影響を与える。報酬は、選択された行動に対応する場合にのみ観測される。環境は完全に観測可能か、または部分的にしか観測不可能な場合がある。これらすべての複雑さを一度に考慮するのは求めすぎかもしれない。さらに、すべての実用的な問題がこの複雑さをすべて示しているわけではない。

その結果、研究者たちは強化学習問題の数多くの特殊なケースを研究してきた。

環境が完全に観測可能である場合、その強化学習問題をマルコフ決定過程（Markov decision process）と呼ぶ。状態が過去の行動に依存しない場合、それを文脈付きバンディット問題（contextual bandit problem）と呼ぶ。状態がなく、最

初は未知の報酬を持つ利用可能な行動の集合のみがある場合、古典的な多腕バンディット問題（multi-armed bandit problem）となる。

4.4 ルーツ

ここまで、機械学習が対処できる問題の小さな部分集合を概観してきた。多様な機械学習の問題に対して、ディープラーニングはそれらを解決するための強力なツールを提供する。多くのディープラーニングの手法は最近の発明であるが、データから学習するという背後にある中核的なアイデアは何世紀にもわたって研究されてきた。実際、人類は古くからデータを分析し、将来の結果を予測したいという欲求を抱いており、多くの自然科学と数学の根底にあるのはこの欲求である。2 つの例として、ヤコブ・ベルヌーイ（1655〜1705）[17] にちなんで名付けられたベルヌーイ分布と、カール・フリードリヒ・ガウス（1777〜1855）[18] によって発見されたガウス分布が挙げられる。ガウスは例えば最小二乗法アルゴリズムを発明し、これは保険計算から医療診断に至るまで、今日でも数多くの問題に使用されている。このようなツールは自然科学における実験的アプローチを強化した。例えば、抵抗器の電流と電圧を関連付けるオームの法則は、線形モデルによって完全に記述される。

17 

18 

中世においてさえ、数学者たちは推定に対する鋭い直感を持っていた。例えば、ヤコブ・ケーベル（1460〜1533）[19] の幾何学の本には、成人男性 16 人の足の長さを平均して、集団の典型的な足の長さを推定する方法が描かれている（Fig. 4.4.1）。

19 

あるグループが教会から出てきたとき、16 人の成人男性が一列に並び、足の長さを測るように頼まれた。これらの測定値の合計を 16 で割り、現在 1 フィートと呼ばれるものの推定値を得たのである。この「アルゴリズム」は後に、形の崩れた足に対処するために改良された。足が最も短い男性と最も長い男性の 2 人を除外し、残りの人だけで平均をとったのである。これは、トリム平均（trimmed mean）推定の最も初期の例の一つである。

統計学はデータの利用と収集によって本格的に飛躍した。その先駆者の一人であるロナルド・フィッシャー（1890〜1962）[20] は、その理論と、遺伝学におけるその応用に大きく貢献した。（線形判別分析などの）彼のアルゴリズムや、（フィッシャー情報行列などの）概念の多くは、現代統計学の基礎において依然として重要な位置を占めている。彼のデータリソースでさえ、永続的な影響を与えた。フィッシャーが 1936 年に公開したアヤメ（Iris）のデータセットは、機械学習アルゴリズムをデモンストレーションするために、現在でも時折使用されている。フィッシャーは優生学の支持者でもあった。このことは、データサイエンスの道徳的に疑わしい使用が、産業や自然科学における生産的な使用と同じくらい長く、そして根強い歴史を持っていることを私たちに思い出させるはずである。

20 

21 

機械学習に対する他の影響は、クロード・シャノン（1916〜2001）[21] の情報理



Fig. 4.4.1 足の長さの推定

論や、アラン・チューリング（1912〜1954）[22] が提案した計算理論からもたらされた。チューリングは有名な論文 *Computing Machinery and Intelligence* (Turing, 1950) の中で、「機械は考えることができるか？」という問いを投げかけた。現在チューリング・テストとして知られているものを説明し、純粋にテキストベースのやり取りに基づいて、人間の評価者が機械からの返答と人間からの返答を区別するのが難しい場合、機械は知的（intelligent）であると見なすことができると彼は提案した。

22

神経科学や心理学からもさらなる影響があった。結局のところ、人間は明らかに知的な振る舞いを示しているからだ。多くの学者が、この能力を説明し、ことによるとリバースエンジニアリングできるのではないかと問いかけてきた。生物学にインスパイアされた最初のアルゴリズムの一つは、ドナルド・ヘッブ（1904〜1985）[23] によって定式化された。彼はその画期的な著書 *The Organization of Behavior* (Hebb, 1949) の中で、ニューロンは正の強化によって学習すると仮定した。これはヘッブの学習則（Hebbian learning rule）として知られるようになった。これらのアイデアは、後のローゼンブラット（Rosenblatt）のパーセプトロン学習アルゴリズムなどの研究に影響を与え、今日のディープラーニングを支える多くの確率的勾配降下法アルゴリズムの基礎を築いた。つまり、ニューラルネットワークにおけるパラメータの優れた設定を得るために、

23

望ましい振る舞いを強化し、望ましくない振る舞いを減少させるというものである。

生物学的なインスピレーションこそが、ニューラルネットワーク（neural networks）という名前を与えたものである。1 世紀以上（1873 年のアレクサンダー・ベイン（Alexander Bain）や 1890 年のジェームズ・シェリントン（James Sherrington）のモデルに遡る）、研究者たちは相互に作用するニューロンのネットワークに似た計算回路を組み立てようと試みてきた。時間の経過とともに生物学の解釈はより文字通りのものではなくなってきたが、名前は定着した。その中核には、今日のほとんどのネットワークに見られるいくつかの重要な原理が存在する：* 線形および非線形処理ユニット（一般に層（layers）と呼ばれる）の交替。* ネットワーク全体のパラメータを一度に調整するための連鎖律（バックプロパゲーション（backpropagation）、誤差逆伝播法としても知られる）の使用。

初期の急速な進歩の後、ニューラルネットワークの研究は 1995 年頃から 2005 年まで停滞した。これには主に 2 つの理由があった。第一に、ネットワークの訓練は計算上非常に高価であること。前世紀の終わり頃にはランダムアクセスメモリ（RAM）は豊富にあったが、計算能力は不足していた。第二に、データセットが比較的小さかったことである。実際、1936 年のフィッシャーの Iris データセットは、依然としてアルゴリズムの有効性をテストするための人気のあるツールであった。6 万の手書き数字からなる MNIST データセットは巨大であると考えられていた。

データと計算資源が不足していたため、カーネル法、決定木、グラフィカルモデルなどの強力な統計ツールが、多くのアプリケーションにおいて経験的に優れていることが証明された。さらに、ニューラルネットワークとは異なり、これらのツールは訓練に何週間も必要とせず、強力な理論的保証を伴う予測可能な結果をもたらした。

4.5 ディープラーニングへの道

ワールド・ワイド・ウェブ（World Wide Web）、オンラインで何億人ものユーザーにサービスを提供する企業の出現、低価格で高品質なセンサーの普及、安価なデータストレージ（クライダーの法則）、そして安価な計算資源（ムーアの法則）のおかげで、大量のデータが利用可能になり、状況は大きく変わった。特に、ディープラーニングにおける計算の展望は、もともとコンピュータゲーム用に設計された GPU の進歩によって革命を起こした。突然、計算上実現不可能に思われていたアルゴリズムやモデルが手の届くところになったのである。これは `tab_intro_decade` に最もよく表れている。

: データセット vs コンピューターのメモリと計算能力

Table 4.5.1: label:tab_intro_decade

年代	データセット	メモリ	1 秒あたりの浮動小数点演算
1970	100 件 (Iris)	1 KB	100 KF (Intel 8080)
1980	1,000 件 (ボストンの住宅価格)	100 KB	1 MF (Intel 80186)
1990	1 万件 (光学文字認識)	10 MB	10 MF (Intel 80486)
2000	1,000 万件 (ウェブページ)	100 MB	1 GF (Intel Core)
2010	100 億件 (広告)	1 GB	1 TF (NVIDIA C2050)
2020	1 兆件 (ソーシャルネットワーク)	100 GB	1 PF (NVIDIA DGX-2)

ランダムアクセスメモリ（RAM）はデータの成長のペースに追いついていないことに注意してほしい。同時に、計算能力の増加はデータセットの成長を上回っている。これは、統計モデルはよりメモリ効率を高める必要がある一方で、増加した計算処理の予算のおかげで、パラメータの最適化に自由に計算サイクルを費やせることを意味する。その結果、機械学習と統計学のスイートスポットは、（一般化）線形モデルやカーネル法からディープニューラルネットワークへと移行した。これは、多層パーセプトロン (McCulloch and Pitts, 1943)、畳み込みニューラルネットワーク (LeCun *et al.*, 1998)、Long Short-Term Memory（LSTM）(Hochreiter and Schmidhuber, 1997)、そして Q 学習 (Watkins and Dayan, 1992) といったディープラーニングの主力手法の多くが、長い間ずっと眠っていた後、過去 10 年の間に本質的に「再発見」された理由の一つでもある。

統計モデル、アプリケーション、そしてアルゴリズムにおける最近の進歩は、種の進化における急速な進歩の瞬間であるカンブリア爆発に例えられることがある。実際、最先端技術は、数十年前のアルゴリズムに利用可能なリソースを適用した単なる結果ではない。以下のアイデアのリストは、過去 10 年間に研究者が驚異的な進歩を遂げるのに役立ったもののほんの一部に過ぎないことに注意してほしい。

- 過学習を軽減するのに役立つ、ドロップアウト（dropout）(Srivastava *et al.*, 2014) などの容量制御のための新しい方法。ここで、訓練中にニューラルネットワーク全体にノイズが注入される (Bishop, 1995)。

- 学習可能なパラメータの数を増やすことなく、システムのメモリと複雑さをどのように増やすかという、1 世紀以上にわたって統計学を悩ませてきた 2 つ目の問題を解決した注意機構（Attention mechanisms）。研究者たちは、学習可能なポインタ構造 (Bahdanau *et al.*, 2014) としか見なせないものを使用することで、エレガントな解決策を見出した。例えば、固定次元の表現における機械翻訳のためにテキストシーケンス全体を記憶する必要はなくなり、保存する必要があるのは、翻訳プロセスの過程にある中間状態へのポインタのみとなった。新しいシーケンスの生成を開始する前にモデルがシーケンス全体を記憶する必要がなくなったため、これにより長いシーケンスに対する精度が大幅に向上した。

- 注意機構のみに基づいて構築された *Transformer* アーキテクチャ (Vaswani *et al.*, 2017) は、優れたスケーリング（scaling）の振る舞いを示した。つまり、データセットのサイズ、モデルのサイズ、そして訓練のための計算量が増加するにつれて、より良い性能を発揮する (Kaplan *et al.*, 2020)。このアーキテクチャは、自然言語処理 (Brown *et al.*, 2020, Devlin *et al.*, 2018)、コンピュータビジョン (Dosovitskiy *et al.*, 2021, Liu *et al.*, 2021)、音声認識 (Gulati *et al.*, 2020)、強化学習 (Chen *et al.*, 2021)、およびグラフ・ニューラル・ネットワーク (Dwivedi and Bresson, 2020) などの幅広い分野で圧倒的な成功を収めている。例えば、テキスト、画像、関節のトルク、ボタンの押下など多様なモダリティで事前学習された単一の Transformer が、Atari をプレイし、画像のキャプションを付け、チャットをし、ロボットを制御できる (Reed *et al.*, 2022)。

- テキストシーケンスの確率をモデル化する言語モデル（language models）は、他のテキストが与えられたときにテキストを予測できる。データ、モデル、計算のスケーリングにより、入力テキストに基づく人間のようなテキスト生成を介して、目的のタスクを実行するための言語モデルの能力がますます解き放たれてきた (Anil *et al.*, 2023, Brown *et al.*, 2020, Chowdhery *et al.*, 2022, Hoffmann *et al.*, 2022, OpenAI, 2023, Rae *et al.*, 2021, Touvron *et al.*, 2023a, Touvron *et al.*, 2023b)。例えば、言語モデルを人間の意図と揃える（アライメント）(Ouyang *et al.*, 2022) ことで、OpenAI のChatGPT[24] は、コードのデバッグやクリエイティブ・ライティングなどの問題を解決するために、ユーザーが対話形式でやり取りできるようにしている。

[24] 

- 例えばメモリネットワーク (Sukhbaatar *et al.*, 2015) やニューラルプログラマとインタプリタ (Reed and De Freitas, 2015) を介した多段階の設計は、統計モデル作成者が推論への反復的アプローチを記述することを可能にした。これらのツールにより、ディープニューラルネットワークの内部状態を繰り返し変更することが可能になり、プロセッサが計算のためにメモリを変更できるのと同じように、推論の連鎖における後続のステップを実行できる。

- 深層生成モデリング（deep generative modeling）における重要な進展は、敵対的生成ネットワーク（generative adversarial networks, GAN）(Goodfellow *et al.*, 2014) の発明であった。従来、密度推定と生成モデルに関する統計的手法は、適切な確率分布と、そこからサンプリングするための（しばしば近似的な）アルゴリズムを見つけることに焦点を当てていた。その結果、これらのアルゴリズムは、統計モデルに固有の柔軟性の欠如によって大きく制限されていた。敵対的生成ネットワークにおける決定的な革新は、サンプラーを微分可能なパラメータを持つ任意のアルゴリズムに置き換えたことであった。その後、識別器（実質的に 2 標本検定）が偽データのデータと実データを見分けられないように調整される。任意のアルゴリズムを使用してデータを生成できる能力により、密度推定は多種多様な技術に開かれた。疾走するシマウマ (Zhu *et al.*, 2017) や実在しない有名人の顔 (Karras *et al.*, 2017) の例は、それぞれこの進歩を証明している。アマチュアの落書き描きでさえ、シーンのレイアウトを説明するスケッチに基づいて、写真のようにリアルな画像を作成できる (Park *et al.*, 2019)。

- さらに、拡散（diffusion）プロセスがデータサンプルにランダムなノイズを徐々に追加する一方で、拡散モデル（diffusion models）(Ho *et al.*, 2020, Sohl-Dickstein *et al.*, 2015) は拡散プロセスを逆転させ、ランダムなノイズからデータサンプルを徐々に構築するノイズ除去プロセスを学習する。テキスト記述に基づいてクリエイティブなアートや画像を生成する DALL-E 2 (Ramesh *et al.*, 2022) や Imagen (Saharia *et al.*, 2022) のような、より最近の深層生成モデルにおいて、拡散モデルは敵対的生成ネットワークに取って代わり始めている。

- 多くの場合、訓練に使用できる大量のデータを処理するには、1 つの GPU では不十分である。過去 10 年間で、並列および分散訓練アルゴリズムを構築する能力は著しく向上した。スケーラブルなアルゴリズムを設計する上での主要な課題の 1 つは、ディープラーニング最適化の主力である確率的勾配降下法が、処理対象のデータとして比較的小さなミニバッチに依存していることである。同時に、小さなバッチは GPU の効率を制限する。したがって、例えば 1 バッチあたり 32 画像のミニバッチサイズで 1,024 個の GPU を使用して訓練すると、合計で約 32,000 画像のミニバッチになる。最初に行われた Li (2017) と、それに続く You *et al.* (2017) および Jia *et al.* (2018) による研究により、サイズは 64,000 の観測値にまで押し上げられ、ImageNet データセット上の ResNet-50 モデルのトレーニング時間は 7 分未満に短縮された。比較すると、当初の訓練時間は数日単位であった。

- 計算を並列化する能力は、強化学習の進歩にも貢献している。これにより、囲碁、Atari ゲーム、スタークラフト、そして環境シミュレータが利用可能な物理シミュレーション（例えば、MuJoCo を使用）などのタスクで、コンピュータが超人的なパフォーマンスを達成するという大きな進歩がもたらされた。AlphaGo におけるそのような成果の説明については、例えば直近の Silver *et al.* (2016) を参照されたい。一言で言えば、十分な（状態、行動、報酬）のタプルが含まれている場合、強化学習は最も効果的に機能する。シミュレーションはそのような手段を提供する。

- ディープラーニングフレームワークは、アイデアの普及に重要な役割を果たしてきた。ニューラルネットワークモデリングのための第 1 世代のオープンソースフレームワークは、Caffe[25] 、Torch[26] 、および Theano[27] から構成されていた。これらのツールを使用して、多くの画期的な論文が執筆された。これらは現在、TensorFlow[28] （多くの場合、高レベル API のKeras[29] を介して使用される）、CNTK[30] 、Caffe 2[31] 、および Apache MXNet[32] に取って代わられている。第 3 世代のフレームワークは、ディープラーニングのためのいわゆる命令型（imperative）ツールから構成されており、この傾向は間違いなくChainer[33] によって火がつけられ、Python の NumPy に似た構文を使用してモデルを記述した。このアイデアは、PyTorch[34] 、MXNet のGluon API[35] 、そしてJAX[36] の両方に採用された。

より良いツールを構築するシステム研究者と、より良いニューラルネットワークを構築する統計モデラーとの間の分業は、物事を大いに単純化した。例えば、線形ロジスティック回帰モデルの訓練はかつて、2014 年のカーネギーメロン大学で機械学習の博士課程の新入生に課すのにふさわしい、難易度の高い宿

題であった。今では、このタスクは 10 行未満のコードで実行できるようになり、あらゆるプログラマーが確実に手の届く範囲にある。

4.6 成功事例

人工知能は、手作業では難しい結果をもたらしてきた長い歴史を持っている。例えば、光学文字認識を使用した郵便物の仕分けシステムは 1990 年代から導入されている。何しろ、これこそが有名な手書き数字の MNIST データセットの出処である。同じことは、銀行預金の小切手の読み取りや、申込者の信用力（クレジットスコア）のスコアリングにも当てはまる。金融取引における不正の有無は自動的にチェックされる。これは、PayPal、Stripe、AliPay、WeChat、Apple、Visa、MasterCard など、多くの E コマース決済システムのバックボーンを形成している。チェスのためのコンピュータプログラムは、何十年もの間競争力を維持してきた。機械学習は、インターネット上での検索、推薦、パーソナライゼーション、およびランキングに情報を提供している。言い換えれば、機械学習はしばしば目立たないものの、どこにでも浸透しているのである。

AI が脚光を浴びるようになったのはつい最近のことであり、それは主に、以前は解決不可能と考えられていた、消費者と直接関係する問題の解決策によるものである。そのような進歩の多くは、ディープラーニングによるものである。

- Apple の Siri、Amazon の Alexa、Google アシスタントなどのインテリジェント・アシスタントは、音声による要求にある程度正確に応答できる。これには、照明のスイッチを入れるといった単純な仕事から、美容院の予約手配や電話サポートの対話などのより複雑なタスクまでが含まれる。これはおそらく、AI が私たちの生活に影響を与えていることの最も目に見える兆候である。

- デジタルアシスタントの重要な要素は、音声を正確に認識する能力である。そのようなシステムの精度は徐々に向上し、特定のアプリケーションでは人間と同等のレベルに達している (Xiong *et al.*, 2018)。

- 物体認識もまた大きく進歩した。写真の中の物体を特定することは、2010 年当時はかなり困難なタスクであった。ImageNet のベンチマークにおいて、NEC の研究所（NEC Labs）とイリノイ大学アーバナ・シャンペーン校の研究者たちは、Top-5 エラー率 28% を達成した (Lin *et al.*, 2010)。2017 年までに、このエラー率は 2.25% まで減少した (Hu *et al.*, 2018)。同様に、鳥のさえずりの特定や皮膚がんの診断においても、驚くべき結果が達成されている。

- ゲームにおける優れた能力は、人間の能力の測定基準を提供するために使用されていた。時間差の強化学習を使用してバックギャモンをプレイするプ

ログラムである TD-Gammon から始まり、アルゴリズムと計算機能の進歩により、幅広いアプリケーションのためのアルゴリズムが生まれた。バックギャモンと比較して、チェスははるかに複雑な状態空間と行動の集合を持っている。DeepBlue は、大規模な並列処理、専用ハードウェア、およびゲームツリーを通じた効率的な検索を使用して、ガルリ・カスパロフ（Garry Kasparov）を破った (Campbell *et al.*, 2002)。巨大な状態空間のために、囲碁はさらに困難である。AlphaGo は、ディープラーニングとモンテカルロ木探索を組み合わせて使用し、2015 年に人間と同等のレベルに到達した (Silver *et al.*, 2016)。ポーカーにおける課題は、状態空間が大きく、部分的にしか観測できないことであった（相手のカードがわからない）。Libratus は、効率的に構造化された戦略を使用して、ポーカーで人間の性能を超えた (Brown and Sandholm, 2017)。

- AI の進歩を示すもう一つの指標は、自動運転車の登場である。完全な自律性はまだ手の届くところにはないが、テスラ、NVIDIA、Waymo などの企業が部分的な自律性を可能にする製品を出荷しており、この方向で素晴らしい進歩が見られている。完全な自律性をこれほどまでに困難にしているのは、適切な運転には、認識し、推論し、システムにルールを組み込む能力が必要だからである。現在、ディープラーニングは主にこれらの問題の視覚的な部分で使用されている。残りの大部分はエンジニアによって大幅に調整されている。

これらは、機械学習の重要な応用のほんの一部に過ぎない。例えば、ロボット工学、物流、計算生物学、素粒子物理学、および天文学は、少なくとも部分的には、その最も印象的な最近の進歩を機械学習に負っている。このように機械学習はエンジニアや科学者にとってユビキタスなツールになりつつある。

非技術的な記事では、来るべき AI の脅威（アポカリプス）や、シンギュラリティ（singularity, 技術的特異点）の妥当性に関する疑問が頻繁に提起されている。なぜか機械学習システムが意識（sentient）を持ち、プログラマーから独立して、人間の生活に直接影響を与える意思決定を行うようになるのではないかという恐れである。ある程度、AI はすでに人間の生活に直接影響を与えている。クレジットスコア（信用力）は自動的に精査され、自動操縦は主に車両をナビゲートし、保釈を許可するかどうかの決定は統計データを入力として使用している。もっと軽いところでは、Alexa にコーヒーマシンの電源を入れるように頼むことができる。

幸いなことに、私たちは人間である創造者を意図的に操作できるような、意識を持つ AI システムからはほど遠い状態にある。第一に、AI システムは特定の、目標指向の（goal-oriented）方法で設計、訓練、および展開される。その振る舞いは一般的な知能と錯覚させるかもしれないが、設計の根底にあるのはルール、ヒューリスティクス、そして統計モデルの組み合わせである。第二に、現在、自己を改善し、自己について推論し、一般的なタスクを解決しようとしながら独自のアーキテクチャを変更、拡張、改善できる汎用人工知能（artificial general intelligence）のためのツールは全く存在しない。

はるかに差し迫った懸念は、私たちの日常生活で AI がどのように使用されているかということである。現在人間によって果たされている多くの定型的なタスクが自動化される可能性は高く、またそうなるだろう。農場ロボットは有機農業者のコストを削減する可能性が高いが、収穫作業も自動化する。多くの国で単純労働が多くの雇用を提供しているため、産業革命のこの段階は社会の広い層に深刻な結果をもたらす可能性がある。さらに、統計モデルを不注意に適用すると、人種、性別、または年齢の偏見（バイアス）につながる可能性があり、重大な意思決定を左右するように自動化された場合、手続きの公平性に関する合理的な懸念を呼び起こす。これらのアルゴリズムが注意深く使用されることを確実にすることが重要である。今日私たちが知っていることについて踏まえると、これは悪意のある超知能が人類を滅ぼす潜在的な可能性よりもはるかに差し迫った懸念として私たちに突きつけられている。

4.7 ディープラーニングの本質

これまで、機械学習について大まかに説明してきた。ディープラーニングは、多層のニューラルネットワークに基づいたモデルに関わる機械学習の部分集合である。そのモデルが多くの層（layers）の変換を学習するというまさにその意味において、それは深い（deep）のである。これは狭い意味に聞こえるかもしれないが、ディープラーニングは目も眩むような数のモデル、技術、問題定式化、および応用の配列を生み出してきた。深さの利点を説明するために、多くの直感的な解釈が開発されてきた。間違いなく、すべての機械学習は多くの計算の層を持ち、最初の一層は特徴量処理のステップから構成されている。ディープラーニングが異なるのは、表現の多くの各層で学習される操作が、データから結合的に（jointly）学習される点である。

生の音声信号や画像の生のピクセル値からの学習、あるいは任意の長さの文と外国語のそれとの間のマッピングなど、これまでに議論してきた問題はディープラーニングが優れており、従来の手法が行き詰まる問題である。これらの多層モデルは、低レベルの知覚データに、これまでのツールではできなかった方法で対処できることが判明している。ディープラーニング手法における最も重要な共通点は、おそらくエンドツーエンドの訓練（end-to-end training）である。つまり、個別に調整されたコンポーネントに基づいてシステムを構築するのではなく、システムを構築し、それらの性能を結合的に調整するのである。例えば、コンピュータビジョンにおいて科学者たちはかつて、特徴量エンジニアリング（feature engineering）のプロセスを、機械学習モデルを構築するプロセスから分離していた。Canny エッジ検出器 (Canny, 1987) と Lowe の SIFT 特徴量抽出器 (Lowe, 2004) は、画像を特徴量ベクトルにマッピングするためのアルゴリズムとして 10 年以上にわたって君臨した。過去の時代には、これらの問題に機械学習を適用する際の重要な部分は、浅いモデル（shallow models）に適した形式にデータを変換するための手動で設計された方法を考え出すことであった。残念ながら、アルゴリズムによって自動的に実行される何百万もの選択肢に

わたる一貫した評価と比較すると、人間の創意工夫によって達成できることには限りがある。ディープラーニングが主導権を握ったとき、これらの特徴量抽出器は、優れた精度をもたらす自動的に調整されたフィルターに置き換えられた。

したがって、ディープラーニングの重要な利点の 1 つは、従来の機械学習パイプラインの最後の浅いモデルだけでなく、特徴量エンジニアリングの労働集約的なプロセスを置き換えることである。さらに、ドメイン固有の前処理の大部分を置き換えることで、ディープラーニングは以前にコンピュータビジョン、音声認識、自然言語処理、医療情報学、およびその他の応用分野を隔てていた境界の多くを取り除き、多様な問題に取り組むための統一されたツール群を提供した。

エンドツーエンドの訓練を超えて、私たちはパラメトリックな統計的記述から完全にノンパラメトリックなモデルへの移行を経験している。データが少ない場合、有用なモデルを得るためには現実に関する単純化された仮定に頼る必要がある。データが豊富な場合、これらはデータによりよく適合するノンパラメトリックモデルに置き換えることができる。ある意味で、これは前世紀半ばにコンピュータの普及に伴って物理学が経験した進歩を反映している。電子がどのように振る舞うかのパラメトリック近似を手動で解くのではなく、関連する偏微分方程式の数値シミュレーションに頼ることができるようになった。これは、しばしば解釈を犠牲にするものの、はるかに正確なモデルをもたらした。

これまでの研究とのもう一つの違いは、非凸（nonconvex）の非線形最適化問題に対処し、次善の（suboptimal）解決策を受け入れること、そして証明する前に試してみるという意欲（willingness）である。統計問題に対処する際のこの新たな経験主義は、才能ある人材の急速な流入と相まって、数十年存在していたツールを変更・再発明するという犠牲を伴うことも多いが、実用的なアルゴリズムの開発において急速な進歩をもたらした。

最後に、ディープラーニングのコミュニティは、学術的境界と企業的境界を越えてツールを共有し、多くの優れたライブラリ、統計モデル、そして訓練済みのネットワークをオープンソースとしてリリースしていることに誇りを持っている。私たちが本書を構成するノートブックを配布および使用のために無料で提供しているのは、このような精神に基づいているからである。ディープラーニングの学習を希望する誰もがアクセスできる障壁を下げるために懸命に取り組んでおり、読者の皆様がここから恩恵を受けることを願っている。

4.8 まとめ

機械学習は、コンピュータシステムが経験（多くの場合データ）を活用して特定のタスクの性能を向上させる方法を研究する学問である。統計学、データマイニング、最適化のアイデアを組み合わせている。多くの場合、AI ソリュー

ションを実装する手段として使用される。機械学習のクラスとして、表現学習（representational learning）は、データを表現する適切な方法を自動的に見つける方法に焦点を当てている。多層の変換の学習を通じたマルチレベルの表現学習と見なされるディープラーニングは、従来の機械学習パイプラインの最後の浅いモデルだけでなく、特徴量エンジニアリングの労働集約的なプロセスも置き換える。ディープラーニングの最近の進歩の多くは、安価なセンサーやインターネット規模のアプリケーションから生じる豊富なデータと、主に GPU による計算能力の著しい進歩によって引き起こされた。さらに、効率的なディープラーニングフレームワークの利用可能性は、システム全体の最適化の設計と実装を大幅に容易にし、これが高いパフォーマンスを得るための重要な要素となっている。

4.9 練習問題

1. 現在書いているコードのうち、どの部分が「学習される」可能性があるか？すなわち、学習によって改善され、コード内で行われている設計上の選択を自動的に決定できる可能性があるか？コードにヒューリスティックな設計上の選択が含まれているか？望ましい振る舞いを学習するためにはどのようなデータが必要か？
2. 直面している問題のうち、その解決に向けた多くのデータ例があるにもかかわらず、それらを自動化する特定の方法がないものは何か？これらはディープラーニングを使用するための有力な候補となる可能性がある。
3. アルゴリズム、データ、および計算能力の関係を説明せよ。データの特性と現在利用可能な計算リソースは、様々なアルゴリズムの妥当性にどのように影響するか？
4. 現在エンドツーエンドの訓練がデフォルトのアプローチではないが、おそらく有用であろう設定をいくつか挙げよ。

5 前提知識

ディープラーニングの世界に飛び込む準備として、いくつかの基本的なスキルが必要になる。(i) データを保存し操作するための技術；(ii) さまざまなソースからデータを取り込み、前処理するためのライブラリ；(iii) 高次元データ要素に適用する基本的な線形代数演算の知識；(iv) 損失関数を減少させるために各パラメータをどの方向に調整すべきかを判断するのに十分な微積分の知識；(v) 自動的に導関数を計算する能力。これにより、今学んだ微積分の多くを忘れてしまっても大丈夫である。(vi) 不確実性の下で推論するための私たちの主要な言語である確率論の基本的な流暢さ；ドキュメントから答えを見つけるための適性である。

要するに、この章では本書の 大部分の技術的内容を追っていくために必要となる基礎を、手早く紹介する。

5.1 データ操作

何らかの実用的な処理を行うには、データを保存し、操作するための方法が必要になる。一般的に、データに対して行うべき重要なことは 2 つある。(i) データを取得すること、そして (ii) 取得したデータをコンピュータに取り込んだ後に処理することである。データを保存する方法がなければ、データを取得しても意味がない。そこでまずは、テンソルとも呼ばれる n 次元配列の操作に慣れていきよう。NumPy のような科学技術計算パッケージをすでに知っているなら、これは簡単だろう。現代のあらゆる深層学習フレームワークにおいて、テンソルクラス（MXNet では `ndarray`、PyTorch と TensorFlow では `Tensor`）は NumPy の `ndarray` に似ているが、いくつかの強力な機能が追加されている。第一に、テンソルクラスは自動微分をサポートしている。第二に、NumPy が CPU 上でしか動作しないのに対し、テンソルは GPU を活用して数値計算を高速化できる。これらの特性により、ニューラルネットワークは実装しやすく、かつ高速に実行できるようになる。

5.1.1 はじめに

まず、PyTorch ライブラリをインポートする。パッケージ名は `torch` であることに注意しよ。

```
import torch
```

```
from mxnet import np, npx
npx.set_np()
```

```
import jax
from jax import numpy as jnp
```

```
import tensorflow as tf
```

テンソルは、数値の（場合によっては多次元の）配列を表す。
1 次元の場合、すなわちデータに 1 つの軸しか必要ない場合、テンソルは ベクトルと呼ばれる。2 つの軸を持つテンソルは 行列と呼ばれる。$k > 2$ の軸を持つ場合は、特別な名前は使わず、単に k-階テンソルと呼ぶ。

PyTorch には、値をあらかじめ入れた新しいテンソルを作成するためのさまざまな関数がある。たとえば arange(n) を呼び出すと、0（含む）から n（含まない）までの等間隔の値を持つベクトルを作成できる。デフォルトでは、間隔の大きさは 1 である。特に指定しない限り、新しいテンソルは主記憶に保存され、CPU ベースの計算に割り当てられる。

```
x = torch.arange(12, dtype=torch.float32)
x
```

```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

```
x = np.arange(12)
x
```

```
[07:03:50] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

```
x = jnp.arange(12)
x
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
Array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11], dtype=int32)
```

```
x = tf.range(12, dtype=tf.float32)
x
```

```
<tf.Tensor: shape=(12,), dtype=float32, numpy=
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.],
      dtype=float32)>
```

これらの各値はテンソルの 要素と呼ばれる。テンソル x には 12 個の要素がある。テンソル内の要素数の総数は numel メソッドで確認できる。

```
x.numel()
```

```
12
```

```
x.size
```

```
12
```

```
x.size
```

```
12
```

```
tf.size(x)
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=12>
```

（テンソルの shape（各軸に沿った長さ））は shape 属性を調べることで取得できる。ここではベクトルを扱っているので、shape には 1 つの要素しか含まれず、サイズと同じである。

```
x.shape
```

```
torch.Size([12])
```

reshape を呼び出すことで、サイズや値を変えずにテンソルの形状を変更できる。たとえば、形状が (12,) のベクトル x を、形状が (3, 4) の行列 X に変換できる。この新しいテンソルはすべての要素を保持したまま、それらを行列として再配置する。ベクトルの要素は 1 行ずつ並んでいるので、x[3] == X[0, 3] となることに注意しよ。

```
X = x.reshape(3, 4)
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

```
X = x.reshape(3, 4)
X
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

```
X = x.reshape(3, 4)
X
```

```
Array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]], dtype=int32)
```

```
X = tf.reshape(x, (3, 4))
X
```

```
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]], dtype=float32)>
```

reshape に対して各 shape 成分をすべて明示するのは冗長であることに注意しよ。テンソルのサイズはすでに分かっているので、他の成分が分かれば shape の 1 成分を求められる。たとえば、サイズが n で目標の shape が (h, w) のテンソルでは、$w = n/h$ であることが分かりる。shape の成分を自動的に推定させるには、自動推定されるべき成分に -1 を指定する。ここでは、x.reshape(3, 4) の代わりに x.reshape(-1, 4) や x.reshape(3, -1) と書いても同じである。

実務では、すべて 0 や 1 で初期化されたテンソルを扱うことがよくある。すべての要素が 0 に設定されたテンソルを構築する（[STRIKEOUT: または 1]）には、zeros 関数を使って shape を (2, 3, 4) にする。

```
torch.zeros((2, 3, 4))
```

```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

```
np.zeros((2, 3, 4))
```

```
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]])
```

```
jnp.zeros((2, 3, 4))
```

```
Array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]], dtype=float32)
```

```
tf.zeros((2, 3, 4))
```

```
<tf.Tensor: shape=(2, 3, 4), dtype=float32, numpy=
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]], dtype=float32)>
```

同様に、ones を呼び出すことで、すべて 1 のテンソルを作成できる。

```
torch.ones((2, 3, 4))
```

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

```
np.ones((2, 3, 4))
```

```
array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

```
jnp.ones((2, 3, 4))
```

```
Array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]], dtype=float32)
```

```
tf.ones((2, 3, 4))
```

```
<tf.Tensor: shape=(2, 3, 4), dtype=float32, numpy=
array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]], dtype=float32)>
```

しばしば、与えられた確率分布から各要素をランダムに（かつ独立に）サンプリングする必要がある。たとえば、ニューラルネットワークのパラメータはしばしばランダムに初期化される。次のコード片は、平均 0、標準偏差 1 の標準ガウス（正規）分布から要素を抽出したテンソルを作成する。

```
torch.randn(3, 4)
```

```
tensor([[ 1.5124,  0.2588,  0.5924,  0.2041],
        [-0.9516,  1.5309,  0.7024,  0.0892],
        [ 1.4337, -0.3017, -1.4316,  0.5385]])
```

```
np.random.normal(0, 1, size=(3, 4))
```

```
array([[ 2.2122064 ,  1.1630787 ,  0.7740038 ,  0.4838046 ],
       [ 1.0434403 ,  0.29956347,  1.1839255 ,  0.15302546],
       [ 1.8917114 , -1.1688148 , -1.2347414 ,  1.5580711 ]])
```

```
# Any call of a random function in JAX requires a key to be
# specified, feeding the same key to a random function will
# always result in the same sample being generated
jax.random.normal(jax.random.PRNGKey(0), (3, 4))
```

```
Array([[ 1.1901639 , -1.0996888 ,  0.44367844,  0.5984697 ],
       [-0.39189556,  0.69261974,  0.46018356, -2.068578  ],
       [-0.21438177, -0.9898306 , -0.6789304 ,  0.27362573]],      ␣
↪dtype=float32)
```

```
tf.random.normal(shape=[3, 4])
```

```
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[ 0.01757281, -1.1219722 ,  0.82192016, -0.35135165],
       [-0.21407793, -0.0028088 , -1.709675  , -1.2871331 ],
       [ 0.56736195,  1.1376742 , -0.1567428 ,  1.146607  ]],
      dtype=float32)>
```

最後に、数値リテラルを含む（場合によっては入れ子になった）Python のリストを与えることで、各要素の正確な値を指定してテンソルを構築できる。ここでは、リストのリストを使って行列を構築している。最外側のリストが軸 0 に対応し、内側のリストが軸 1 に対応する。

```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

```
np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
array([[2., 1., 4., 3.],
```

(continues on next page)

(continued from previous page)

```
       [1., 2., 3., 4.],
       [4., 3., 2., 1.]])
```

```
jnp.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
Array([[2, 1, 4, 3],
       [1, 2, 3, 4],
       [4, 3, 2, 1]], dtype=int32)
```

```
tf.constant([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
<tf.Tensor: shape=(3, 4), dtype=int32, numpy=
array([[2, 1, 4, 3],
       [1, 2, 3, 4],
       [4, 3, 2, 1]], dtype=int32)>
```

5.1.2 インデックス付けとスライシング

Python のリストと同様に、インデックス付け（0 から始まる）によってテンソルの要素にアクセスできる。リストの末尾からの位置に基づいて要素にアクセスするには、負のインデックスを使う。さらに、スライシング（たとえば `X[start:stop]`）によってインデックスの範囲全体にアクセスできる。このとき返される値には最初のインデックス（`start`）は含まれるが、最後のインデックス（`stop`）は含まれない。最後に、k-階テンソルに対して 1 つのインデックス（またはスライス）だけを指定した場合、それは軸 0 に沿って適用される。したがって、次のコードでは、`[-1]` は最後の行を選択し、`[1:3]` は 2 行目と 3 行目を選択する。

```
X[-1], X[1:3]
```

```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

読み取るだけでなく、（**インデックスを指定して行列の要素を書き込むこともできる**。）

```
X[1, 2] = 17
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
```

(continues on next page)

(continued from previous page)

```
        [ 4.,  5., 17.,  7.],
        [ 8.,  9., 10., 11.]])
```

```
X[1, 2] = 17
X
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5., 17.,  7.],
       [ 8.,  9., 10., 11.]])
```

```
# JAX arrays are immutable. jax.numpy.ndarray.at index
# update operators create a new array with the corresponding
# modifications made
X_new_1 = X.at[1, 2].set(17)
X_new_1
```

```
Array([[ 0,  1,  2,  3],
       [ 4,  5, 17,  7],
       [ 8,  9, 10, 11]], dtype=int32)
```

```
X_var = tf.Variable(X)
X_var[1, 2].assign(9)
X_var
```

```
<tf.Variable 'Variable:0' shape=(3, 4) dtype=float32, numpy=
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  9.,  7.],
       [ 8.,  9., 10., 11.]], dtype=float32)>
```

複数の要素に同じ値を代入したい場合は、代入操作の左辺でインデックス付けを行う。たとえば、`[:2, :]` は 1 行目と 2 行目にアクセスし、`:` は軸 1（列）に沿ったすべての要素を取りる。ここでは行列についてインデックス付けを説明したが、これはベクトルや 2 次元より高いテンソルにも同様に使える。

```
X[:2, :] = 12
X
```

```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

```
X[:2, :] = 12
X
```

```
array([[12., 12., 12., 12.],
       [12., 12., 12., 12.],
       [ 8.,  9., 10., 11.]])
```

```
X_new_2 = X_new_1.at[:2, :].set(12)
X_new_2
```

```
Array([[12, 12, 12, 12],
       [12, 12, 12, 12],
       [ 8,  9, 10, 11]], dtype=int32)
```

```
X_var = tf.Variable(X)
X_var[:2, :].assign(tf.ones(X_var[:2,:].shape, dtype=tf.float32) * 12)
X_var
```

```
<tf.Variable 'Variable:0' shape=(3, 4) dtype=float32, numpy=
array([[12., 12., 12., 12.],
       [12., 12., 12., 12.],
       [ 8.,  9., 10., 11.]], dtype=float32)>
```

5.1.3 演算

テンソルの構築方法と、その要素の読み書き方法が分かったので、さまざまな数学演算を使ってテンソルを操作できるようになる。中でも最も有用なのは 要素ごとの演算である。これは、標準的なスカラー演算をテンソルの各要素に適用する。2 つのテンソルを入力に取る関数では、要素ごとの演算は対応する要素の各組に標準的な二項演算子を適用する。スカラーをスカラーへ写す任意の関数から、要素ごとの関数を作ることができる。

数学記法では、このような 単項スカラー演算子（1 つの入力を取る）を、シグネチャ $f : \mathbb{R} \to \mathbb{R}$ で表す。これは、関数が任意の実数を別の実数へ写すことを意味する。e^x のような単項演算子を含む、ほとんどの標準的な演算子は要素ごとに適用できる。

```
torch.exp(x)
```

```
tensor([162754.7969, 162754.7969, 162754.7969, 162754.7969, 162754.7969,
        162754.7969, 162754.7969, 162754.7969,   2980.9580,   8103.0840,
         22026.4648,  59874.1406])
```

```
np.exp(x)
```

```
array([1.0000000e+00, 2.7182817e+00, 7.3890562e+00, 2.0085537e+01,
       5.4598148e+01, 1.4841316e+02, 4.0342880e+02, 1.0966332e+03,
       2.9809580e+03, 8.1030840e+03, 2.2026465e+04, 5.9874141e+04])
```

```
jnp.exp(x)
```

```
Array([1.0000000e+00, 2.7182817e+00, 7.3890562e+00, 2.0085537e+01,
       5.4598152e+01, 1.4841316e+02, 4.0342880e+02, 1.0966332e+03,
       2.9809580e+03, 8.1030840e+03, 2.2026465e+04, 5.9874141e+04],      
↪dtype=float32)
```

```
tf.exp(x)
```

```
<tf.Tensor: shape=(12,), dtype=float32, numpy=
array([1.0000000e+00, 2.7182817e+00, 7.3890562e+00, 2.0085537e+01,
       5.4598148e+01, 1.4841316e+02, 4.0342877e+02, 1.0966332e+03,
       2.9809580e+03, 8.1030835e+03, 2.2026465e+04, 5.9874141e+04],
      dtype=float32)>
```

同様に、実数の組を 1 つの実数へ写す 二項スカラー演算子は、シグネチャ $f : \mathbb{R}, \mathbb{R} \to \mathbb{R}$ で表す。任意の 2 つのベクトル $\mathbf{u}$ と $\mathbf{v}$ が同じ *shape* を持ち、二項演算子 f が与えられたとき、各 i について $c_i \leftarrow f(u_i, v_i)$ と置くことで、ベクトル $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ を生成できる。ここで、c_i, u_i, v_i はそれぞれベクトル $\mathbf{c}, \mathbf{u}, \mathbf{v}$ の i-番目の要素である。ここでは、スカラー関数を要素ごとのベクトル演算へ 持ち上げることで、ベクトル値の $F : \mathbb{R}^d, \mathbb{R}^d \to \mathbb{R}^d$ を得ている。加算（+）、減算（-）、乗算（*）、除算（/）、べき乗（**）といった一般的な標準算術演算子は、任意の shape を持つ同じ shape のテンソルに対して、すべて 持ち上げられ、要素ごとの演算として使えるようになっている。

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
```

```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1.,  4., 16., 64.]))
```

```
x = np.array([1, 2, 4, 8])
y = np.array([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
```

```
(array([ 3.,  4.,  6., 10.]),
 array([-1.,  0.,  2.,  6.]),
 array([ 2.,  4.,  8., 16.]),
 array([0.5, 1. , 2. , 4. ]),
 array([ 1.,  4., 16., 64.]))
```

```
x = jnp.array([1.0, 2, 4, 8])
y = jnp.array([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
```

```
(Array([ 3.,  4.,  6., 10.], dtype=float32),
 Array([-1.,  0.,  2.,  6.], dtype=float32),
 Array([ 2.,  4.,  8., 16.], dtype=float32),
 Array([0.5, 1. , 2. , 4. ], dtype=float32),
 Array([ 1.,  4., 16., 64.], dtype=float32))
```

```
x = tf.constant([1.0, 2, 4, 8])
y = tf.constant([2.0, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
```

```
(<tf.Tensor: shape=(4,), dtype=float32, numpy=array([ 3.,  4.,  6., 10.],␣
↪dtype=float32)>,
 <tf.Tensor: shape=(4,), dtype=float32, numpy=array([-1.,  0.,  2.,  6.],␣
↪dtype=float32)>,
 <tf.Tensor: shape=(4,), dtype=float32, numpy=array([ 2.,  4.,  8., 16.],␣
↪dtype=float32)>,
 <tf.Tensor: shape=(4,), dtype=float32, numpy=array([0.5, 1. , 2. , 4. ],␣
↪dtype=float32)>,
 <tf.Tensor: shape=(4,), dtype=float32, numpy=array([ 1.,  4., 16., 64.],␣
↪dtype=float32)>)
```

要素ごとの計算に加えて、内積や行列積のような線形代数演算も実行できる。これらについては Section 5.3 で詳しく説明する。

また、複数のテンソルを連結することもでき、端から端へつなげてより大きなテンソルを作れる。そのためには、テンソルのリストを与え、どの軸に沿って連結するかをシステムに指定するだけである。以下の例では、2 つの行列を列（軸 1）ではなく行（軸 0）に沿って連結したときに何が起こるかを示している。最初の出力の軸 0 の長さ（6）は、2 つの入力テンソルの軸 0 の長さ（3+3）の和であり、2 つ目の出力の軸 1 の長さ（8）は、2 つの入力テンソルの軸 1 の長さ（4 + 4）の和であることが分かりる。

```
X = torch.arange(12, dtype=torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [ 2.,  1.,  4.,  3.],
         [ 1.,  2.,  3.,  4.],
         [ 4.,  3.,  2.,  1.]]),
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
         [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
         [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

```
X = np.arange(12).reshape(3, 4)
Y = np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
np.concatenate([X, Y], axis=0), np.concatenate([X, Y], axis=1)
```

```
(array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [ 2.,  1.,  4.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 4.,  3.,  2.,  1.]]),
 array([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

```
X = jnp.arange(12, dtype=jnp.float32).reshape((3, 4))
Y = jnp.array([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
jnp.concatenate((X, Y), axis=0), jnp.concatenate((X, Y), axis=1)
```

```
(Array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [ 2.,  1.,  4.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 4.,  3.,  2.,  1.]], dtype=float32),
 Array([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]], dtype=float32))
```

```
X = tf.reshape(tf.range(12, dtype=tf.float32), (3, 4))
Y = tf.constant([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
tf.concat([X, Y], axis=0), tf.concat([X, Y], axis=1)
```

```
(<tf.Tensor: shape=(6, 4), dtype=float32, numpy=
 array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [ 2.,  1.,  4.,  3.],
        [ 1.,  2.,  3.,  4.],
```

(continues on next page)

(continued from previous page)

```
       [ 4.,  3.,  2.,  1.]], dtype=float32)>,
 <tf.Tensor: shape=(3, 8), dtype=float32, numpy=
 array([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]], dtype=float32)>)
```

ときには、論理式を通じて二値テンソルを構築したいことがある。`X == Y` を例にしよう。各位置 i, j について、`X[i, j]` と `Y[i, j]` が等しければ、結果の対応する要素は 1 になり、そうでなければ 0 になる。

```
X == Y
```

```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

テンソルのすべての要素を合計すると、要素が 1 つだけのテンソルが得られる。

```
X.sum()
```

```
tensor(66.)
```

```
X.sum()
```

```
array(66.)
```

```
X.sum()
```

```
Array(66., dtype=float32)
```

```
tf.reduce_sum(X)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=66.0>
```

5.1.4 ブロードキャスティング

ここまでで、同じ shape を持つ 2 つのテンソルに対して要素ごとの二項演算を行う方法を学んだ。ある条件下では、shape が異なっていても、ブロードキャスティング機構を呼び出すことで要素ごとの二項演算を実行できる。ブロードキ

ャスティングは次の 2 段階の手順に従いる。(i) 長さ 1 の軸に沿って要素をコピーすることで、一方または両方の配列を拡張し、この変換後に 2 つのテンソルが同じ shape になるようにする。(ii) 得られた配列に対して要素ごとの演算を行う。

```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

```
a = np.arange(3).reshape(3, 1)
b = np.arange(2).reshape(1, 2)
a, b
```

```
(array([[0.],
        [1.],
        [2.]]),
 array([[0., 1.]]))
```

```
a = jnp.arange(3).reshape((3, 1))
b = jnp.arange(2).reshape((1, 2))
a, b
```

```
(Array([[0],
        [1],
        [2]], dtype=int32),
 Array([[0, 1]], dtype=int32))
```

```
a = tf.reshape(tf.range(3), (3, 1))
b = tf.reshape(tf.range(2), (1, 2))
a, b
```

```
(<tf.Tensor: shape=(3, 1), dtype=int32, numpy=
 array([[0],
        [1],
        [2]], dtype=int32)>,
 <tf.Tensor: shape=(1, 2), dtype=int32, numpy=array([[0, 1]], dtype=int32)>)
```

a と b はそれぞれ 3×1 と 1×2 の行列なので、shape は一致しない。ブロードキャスティングでは、行列 a を列方向に、行列 b を行方向に複製してから要素ごとに加算することで、より大きな 3×2 行列を生成する。

```
a + b
```

```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

5.1.5 メモリの節約

演算を実行すると、結果を保持するために新しいメモリが割り当てられることがある。たとえば `Y = X + Y` と書くと、`Y` が以前指していたテンソルへの参照を外し、代わりに `Y` を新しく割り当てられたメモリに向ける。この問題は Python の `id()` 関数で確認できる。これは参照先オブジェクトのメモリアドレスを正確に返す。`Y = Y + X` を実行した後、`id(Y)` が別の場所を指していることに注意しよ。これは、Python がまず `Y + X` を評価して結果用の新しいメモリを割り当て、その後 `Y` をこの新しいメモリ位置に向けるからである。

```
before = id(Y)
Y = Y + X
id(Y) == before
```

```
False
```

これは 2 つの理由で望ましくない場合がある。第一に、不要なメモリ割り当てを何度も行いたくないからである。機械学習では、しばしば数百メガバイト規模のパラメータを持ち、それらすべてを毎秒何度も更新する。可能な限り、これらの更新は インプレースで行いたいものである。第二に、同じパラメータを複数の変数から参照していることがある。インプレース更新をしない場合、メモリリークを起こしたり、古いパラメータを誤って参照したりしないよう、これらすべての参照を注意深く更新しなければならない。

幸いなことに、インプレース演算の実行は簡単である。スライス記法 `Y[:] = <expression>` を使うことで、演算結果をすでに割り当て済みの配列 `Y` に代入できる。この概念を示すために、`zeros_like` を使って `Y` と同じ shape を持つように初期化したテンソル `Z` の値を上書きする。

```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 139859127252688
id(Z): 139859127252688
```

```
Z = np.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 140413152328000
id(Z): 140413152328000
```

```
# JAX arrays do not allow in-place operations
```

```
Z = tf.Variable(tf.zeros_like(Y))
print('id(Z):', id(Z))
Z.assign(X + Y)
print('id(Z):', id(Z))
```

```
id(Z): 140582589136560
id(Z): 140582589136560
```

X の値が後続の計算で再利用されないなら、`X[:] = X + Y` や `X += Y` を使って演算のメモリオーバーヘッドを減らすこともできる。

```
before = id(X)
X += Y
id(X) == before
```

```
True
```

```
before = id(X)
X += Y
id(X) == before
```

```
True
```

```
A = jax.device_get(X)
B = jax.device_put(A)
type(A), type(B)
```

```
(numpy.ndarray, jaxlib.xla_extension.ArrayImpl)
```

```
@tf.function
def computation(X, Y):
    Z = tf.zeros_like(Y)  # This unused value will be pruned out
```

(continues on next page)

(continued from previous page)

```
    A = X + Y  # Allocations will be reused when no longer needed
    B = A + Y
    C = B + Y
    return C + Y

computation(X, Y)
```

```
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[ 8.,  9., 26., 27.],
       [24., 33., 42., 51.],
       [56., 57., 58., 59.]], dtype=float32)>
```

5.1.6 他の Python オブジェクトへの変換

NumPy テンソル（ndarray）への変換、またはその逆は簡単である。変換結果はメモリを共有しない。この小さな不便さは実は非常に重要である。CPU や GPU 上で演算を行うとき、Python の NumPy パッケージが同じメモリ領域を使って別のことをしたいかどうかを待つために、計算を止めたくはないからである。

NumPy テンソル（ndarray）への変換、またはその逆は簡単である。torch テンソルと NumPy 配列は基盤となるメモリを共有し、一方をインプレース演算で変更すると、もう一方も変更される。

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

```
(numpy.ndarray, torch.Tensor)
```

```
A = X.asnumpy()
B = np.array(A)
type(A), type(B)
```

```
(numpy.ndarray, mxnet.numpy.ndarray)
```

```
a = jnp.array([3.5])
a, a.item(), float(a), int(a)
```

```
(Array([3.5], dtype=float32), 3.5, 3.5, 3)
```

```
A = X.numpy()
```

(continues on next page)

(continued from previous page)

```
B = tf.constant(A)
type(A), type(B)
```

```
(numpy.ndarray, tensorflow.python.framework.ops.EagerTensor)
```

サイズ 1 のテンソルを Python のスカラーに変換するには、`item` 関数や Python の組み込み関数を呼び出せる。

```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

```
(tensor([3.5000]), 3.5, 3.5, 3)
```

```
a = np.array([3.5])
a, a.item(), float(a), int(a)
```

```
(array([3.5]), 3.5, 3.5, 3)
```

```
a = tf.constant([3.5]).numpy()
a, a.item(), float(a), int(a)
```

```
(array([3.5], dtype=float32), 3.5, 3.5, 3)
```

5.1.7 要約

テンソルクラスは、深層学習ライブラリにおいてデータを保存し操作するための主要なインターフェースである。テンソルは、構築ルーチン、インデックス付けとスライシング、基本的な数学演算、ブロードキャスティング、メモリ効率のよい代入、そして他の Python オブジェクトとの相互変換など、さまざまな機能を提供する。

5.1.8 演習

1. この節のコードを実行しよ。条件式 `X == Y` を `X < Y` または `X > Y` に変えて、どのようなテンソルが得られるか確認しよ。

2. ブロードキャスティング機構で要素ごとに作用する 2 つのテンソルを、3 次元テンソルなど別の shape に置き換えよ。結果は期待どおりになるか？

5.2 データ前処理

これまで、私たちはすぐに使えるテンソルとして届く合成データを扱ってきた。

現実世界で深層学習を適用するには、さまざまな形式で保存された雑多なデータを取り込み、必要に応じて前処理を行う必要がある。幸いなことに、*pandas* ライブラリ[37] はその大部分を自動化・簡略化してくれる。この節は pandas の包括的な チュートリアル[38] ではないが、よく使われる基本的なデータ処理についての速習になる。

[37] 

[38] 

5.2.1 データセットの読み込み

カンマ区切り値（CSV）ファイルは、表形式（スプレッドシートのような）データを保存するために広く使われている。

CSV では、各行が 1 つのレコードに対応し、いくつかの（カンマ区切りの）フィールドから構成される。たとえば、“Albert Einstein,March 14 1879,Ulm,Federal polytechnic school,field of gravitational physics”のようなものである。

pandas を使って CSV ファイルを読み込む方法を示すために、ここでは以下で CSV ファイルを作成する `../data/house_tiny.csv` を用いる。

このファイルは住宅データセットを表しており、各行が 1 つの住宅に対応し、列は部屋数（`NumRooms`）、屋根の種類（`RoofType`）、価格（`Price`）を表す。

```
import os

os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('''NumRooms,RoofType,Price
NA,NA,127500
2,NA,106000
4,Slate,178100
NA,NA,140000''')
```

では、pandas をインポートして `read_csv` でデータセットを読み込みよう。

```
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

```
   NumRooms RoofType   Price
```

(continues on next page)

(continued from previous page)

```
0       NaN      NaN  127500
1       2.0      NaN  106000
2       4.0    Slate  178100
3       NaN      NaN  140000
```

5.2.2 データの準備

教師あり学習では、ある一連の 入力値が与えられたときに、指定された 目標値を予測するようにモデルを訓練する。
データセットを処理する最初のステップは、入力値に対応する列と目標値に対応する列を分けることである。
列は名前で選択することも、整数位置に基づくインデックス指定（iloc）で選択することもできる。

お気づきかもしれないが、pandas は CSV の NA の値をすべて、特別な NaN（*not a number*）値に置き換えた。
これは、たとえば“3,,,270000”のように、項目が空欄の場合にも起こりる。
これらは 欠損値と呼ばれ、データサイエンスにおける難所の一つである。あなたのキャリアを通じて向き合い続けることになる厄介な問題と言えるだろう。
文脈に応じて、欠損値は 補完 (imputation) か 削除 (deletion) によって処理される。補完では欠損値を推定値で置き換え、削除では欠損値を含む行または列をデータセットから除外する。

以下に、よく使われる補完のヒューリスティックを示す。
カテゴリ型の入力欄では、NaN を 1 つのカテゴリとして扱うことができる。
RoofType 列は Slate と NaN の値を取るので、pandas はこの列を RoofType_Slate と RoofType_nan の 2 列に変換できる。
屋根の種類が Slate の行では、RoofType_Slate と RoofType_nan の値はそれぞれ 1 と 0 になる。
RoofType の値が欠損している行では、その逆になる。

```
inputs, targets = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

```
   NumRooms  RoofType_Slate  RoofType_nan
0       NaN           False          True
1       2.0           False          True
```

(continues on next page)

(continued from previous page)

```
2       4.0            True         False
3       NaN           False          True
```

数値の欠損値については、よく使われるヒューリスティックとして、NaN の項目を対応する列の平均値で置き換える方法がある。

```
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

```
   NumRooms  RoofType_Slate  RoofType_nan
0       3.0           False          True
1       2.0           False          True
2       4.0            True         False
3       3.0           False          True
```

5.2.3 テンソル形式への変換

これで inputs と targets のすべての項目が数値になったので、テンソルに読み込める（Section 5.1 を思い出してほしい）。

```
import torch

X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(targets.to_numpy(dtype=float))
X, y
```

```
(tensor([[3., 0., 1.],
         [2., 0., 1.],
         [4., 1., 0.],
         [3., 0., 1.]], dtype=torch.float64),
 tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))
```

```
from mxnet import np

X, y = np.array(inputs.to_numpy(dtype=float)), np.array(targets.to_
↪numpy(dtype=float))
X, y
```

```
[07:03:52] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(array([[3., 0., 1.],
        [2., 0., 1.],
        [4., 1., 0.],
        [3., 0., 1.]], dtype=float64),
 array([127500., 106000., 178100., 140000.], dtype=float64))
```

```
from jax import numpy as jnp

X = jnp.array(inputs.to_numpy(dtype=float))
y = jnp.array(targets.to_numpy(dtype=float))
X, y
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
(Array([[3., 0., 1.],
        [2., 0., 1.],
        [4., 1., 0.],
        [3., 0., 1.]], dtype=float32),
 Array([127500., 106000., 178100., 140000.], dtype=float32))
```

```
import tensorflow as tf

X = tf.constant(inputs.to_numpy(dtype=float))
y = tf.constant(targets.to_numpy(dtype=float))
X, y
```

```
(<tf.Tensor: shape=(4, 3), dtype=float64, numpy=
 array([[3., 0., 1.],
        [2., 0., 1.],
        [4., 1., 0.],
        [3., 0., 1.]])>,
 <tf.Tensor: shape=(4,), dtype=float64, numpy=array([127500., 106000., 178100.,
↪ 140000.])>)
```

5.2.4 議論

これで、データ列を分割し、欠損変数を補完し、pandas のデータをテンソルに読み込む方法がわかった。

Section 8.7 では、さらにいくつかのデータ処理スキルを学ぶ。

この速習では話を単純にしたが、データ処理はかなり複雑になりえる。

たとえば、データセットが 1 つの CSV ファイルにまとまっているのではなく、リレーショナルデータベースから抽出された複数のファイルに分散していることがある。

たとえば電子商取引アプリケーションでは、顧客住所はあるテーブルに、購買データは別のテーブルにあるかもしれない。

さらに、実務ではカテゴリ型や数値型以外にも、テキスト文字列、画像、音声データ、点群など、さまざまなデータ型に直面する。

しばしば、データ処理が機械学習パイプラインの最大のボトルネックにならないようにするために、高度なツールや効率的なアルゴリズムが必要になる。

これらの問題は、コンピュータビジョンや自然言語処理に進むと現れてきる。

最後に、データ品質にも注意を払わなければならない。
現実世界のデータセットは、外れ値、センサーによる誤測定、記録ミスなどに悩まされることが多く、データをどのモデルに入れる前にも対処が必要である。
seaborn[39] 、Bokeh[40] 、matplotlib[41] などのデータ可視化ツールは、データを手作業で確認し、どのような問題に対処すべきかについて直感を養うのに役立ちる。

5.2.5 演習

1. たとえば UCI Machine Learning Repository の Abalone などのデータセットを読み込み、その性質を調べてみよう。欠損値を含む割合はどれくらいだろうか。変数のうち、数値型、カテゴリ型、テキスト型の割合はどれくらいだろうか。
2. 列番号ではなく列名によってデータ列をインデックス指定し、選択してみよう。pandas の indexing[42] のドキュメントには、その方法の詳細が載っている。
3. この方法でどれくらい大きなデータセットまで読み込めると思うか。どのような制約があるだろうか。ヒント：データの読み込み時間、表現、処理、メモリ使用量を考えてみよ。自分のノートパソコンで試してみよう。サーバー上で試すとどうなるか。
4. カテゴリ数が非常に多いデータをどのように扱いますか。カテゴリラベルがすべて一意だったらどうだろうか。後者も含めるべきだろうか。
5. pandas の代替として何が考えられますか。ファイルから NumPy テンソルを読み込む方法はどうだろうか。Pillow[43] 、つまり Python Imaging Library も調べてみよう。

5.3 線形代数

ここまでで、データセットをテンソルに読み込み、基本的な数学演算でこれらのテンソルを操作できるようになった。より洗練されたモデルを構築し始めるには、線形代数のいくつかの道具も必要になる。この節では、スカラー演算から始めて行列積へと進みながら、最も重要な概念をやさしく導入する。

```
import torch
```

```
from mxnet import np, npx
npx.set_np()
```

```
from jax import numpy as jnp
```

```
import tensorflow as tf
```

5.3.1 スカラー

数学における基本的な演算の多くは、数値を一つずつ操作することから成り立っている。形式的には、これらの値を スカラーと呼ぶ。たとえば、パロアルトの気温が華氏 72 度であるとする。気温を摂氏に変換したければ、f を 72 に設定して式 $c = \frac{5}{9}(f - 32)$ を評価する。この式では、5、9、32 は定数スカラーである。変数 c と f は一般に未知のスカラーを表す。

スカラーは通常の小文字（たとえば x、y、z）で表し、すべての（連続な）実数値スカラーの空間を $\mathbb{R}$ で表す。手短にするため、空間の厳密な定義は省略する。ここでは、式 $x \in \mathbb{R}$ は x が実数値のスカラーであることを示す形式的な記法だと考えよ。記号 $\in$（「〜に属する」と読みます）は、集合への所属を表す。たとえば、$x, y \in \{0, 1\}$ は、x と y が 0 か 1 しか取れない変数であることを示す。

スカラーは、1 つの要素だけを含むテンソルとして実装される。以下では、2 つのスカラーを代入し、おなじみの加算、乗算、除算、べき乗を行う。

```
x = torch.tensor(3.0)
y = torch.tensor(2.0)

x + y, x * y, x / y, x**y
```

```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

```
x = np.array(3.0)
y = np.array(2.0)

x + y, x * y, x / y, x ** y
```

```
[07:04:15] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(array(5.), array(6.), array(1.5), array(9.))
```

```
x = jnp.array(3.0)
y = jnp.array(2.0)

x + y, x * y, x / y, x**y
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
(Array(5., dtype=float32, weak_type=True),
 Array(6., dtype=float32, weak_type=True),
 Array(1.5, dtype=float32, weak_type=True),
 Array(9., dtype=float32, weak_type=True))
```

```
x = tf.constant(3.0)
y = tf.constant(2.0)

x + y, x * y, x / y, x**y
```

```
(<tf.Tensor: shape=(), dtype=float32, numpy=5.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=6.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=1.5>,
 <tf.Tensor: shape=(), dtype=float32, numpy=9.0>)
```

5.3.2 ベクトル

今のところ、ベクトルはスカラーの固定長配列だと考えて差し支えない。通常の配列と同様に、これらのスカラーをベクトルの 要素と呼ぶ（エントリや 成分とも呼ばれます）。ベクトルが実世界のデータセットの例を表すとき、その値には何らかの現実世界での意味がある。たとえば、ローンのデフォルトリスクを予測するモデルを学習しているなら、各申請者をベクトルに対応付け、その成分は収入、勤務年数、過去のデフォルト回数などの量に対応するかもしれない。心臓発作のリスクを研究しているなら、各ベクトルは患者を表し、その成分は直近のバイタルサイン、コレステロール値、1 日あたりの運動時間などに対応するかもしれない。ベクトルは太字の小文字（たとえば $\mathbf{x}$、$\mathbf{y}$、$\mathbf{z}$）で表す。

ベクトルは 1^{st}-order テンソルとして実装される。一般に、このようなテンソルはメモリ制約の範囲で任意の長さを持てる。注意: Python では、ほとんどのプログラミング言語と同様に、ベクトルの添字は 0 から始まりる。これは ゼロ始まりのインデックス付けとも呼ばれる。一方、線形代数では添字は 1 から始まります（1 始まりのインデックス付け）。

```
x = torch.arange(3)
x
```

```
tensor([0, 1, 2])
```

```
x = np.arange(3)
x
```

```
array([0., 1., 2.])
```

```
x = jnp.arange(3)
x
```

```
Array([0, 1, 2], dtype=int32)
```

```
x = tf.range(3)
x
```

```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([0, 1, 2], dtype=int32)>
```

添字を使ってベクトルの要素を参照できる。たとえば、x_2 は $\mathbf{x}$ の 2 番目の要素を表す。x_2 はスカラーなので、太字にはしない。デフォルトでは、ベクトルは要素を縦に並べて可視化する。

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}. \tag{5.3.1}$$

ここで $x_1, \ldots, x_n$ はベクトルの要素である。後で、こうした 列ベクトルと、要素を横に並べた 行ベクトルを区別する。テンソルの要素にはインデックスでアクセスすることを思い出しよ。

```
x[2]
```

```
tensor(2)
```

ベクトルが n 個の要素を含むことを示すには、$\mathbf{x} \in \mathbb{R}^n$ と書きる。形式的には、n をベクトルの 次元数と呼ぶ。コードでは、これはテンソルの長さに対応する。Python の組み込み関数 `len` で取得できる。

```
len(x)
```

```
3
```

長さは shape 属性でも取得できる。shape は、各軸に沿ったテンソルの長さを示すタプルである。1 つの軸しか持たないテンソルの shape は、1 つの要素だけを持つ。

```
x.shape
```

```
torch.Size([3])
```

しばしば「次元」という語は、軸の数と特定の軸に沿った長さの両方を指すために曖昧に使われる。この混乱を避けるため、軸の数を指すときには 階数を用い、要素数を指すときには 次元数を専ら用いる。

5.3.3 行列

スカラーが 0^{th}-order テンソルであり、ベクトルが 1^{st}-order テンソルであるのと同様に、行列は 2^{nd}-order テンソルである。行列は太字の大文字（たとえば $\mathbf{X}$、$\mathbf{Y}$、$\mathbf{Z}$）で表し、コードでは 2 つの軸を持つテンソルとして表現する。式 $\mathbf{A} \in \mathbb{R}^{m \times n}$ は、行列 $\mathbf{A}$ が $m \times n$ 個の実数値スカラーを含み、m 行 n 列に並んでいることを示す。$m = n$ のとき、その行列は 正方行列と呼ぶ。視覚的には、任意の行列を表として示せる。個々の要素を参照するには、行と列の両方の添字を付ける。たとえば、a_{ij} は $\mathbf{A}$ の i^{th} 行 j^{th} 列に属する値である。

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{5.3.2}$$

コードでは、行列 $\mathbf{A} \in \mathbb{R}^{m \times n}$ を shape が (m, n) の 2^{nd}-order テンソルとして表す。任意の適切なサイズの $m \times n$ テンソルを $m \times n$ 行列に変換できる `reshape` に希望の shape を渡すことで実現できる。

```
A = torch.arange(6).reshape(3, 2)
A
```

```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

```
A = np.arange(6).reshape(3, 2)
A
```

```
array([[0., 1.],
       [2., 3.],
       [4., 5.]])
```

```
A = jnp.arange(6).reshape(3, 2)
A
```

```
Array([[0, 1],
       [2, 3],
       [4, 5]], dtype=int32)
```

```
A = tf.reshape(tf.range(6), (3, 2))
A
```

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[0, 1],
       [2, 3],
       [4, 5]], dtype=int32)>
```

ときには軸を入れ替えたいことがある。行列の行と列を交換した結果を、その転置と呼ぶ。形式的には、行列 $\mathbf{A}$ の転置を $\mathbf{A}^\top$ で表し、$\mathbf{B} = \mathbf{A}^\top$ ならば、すべての i と j について $b_{ij} = a_{ji}$ である。したがって、$m \times n$ 行列の転置は $n \times m$ 行列になる。

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}. \tag{5.3.3}$$

コードでは、任意の行列の転置を次のように取得できる。

```
A.T
```

```
tensor([[0, 2, 4],
        [1, 3, 5]])
```

```
A.T
```

```
array([[0., 2., 4.],
       [1., 3., 5.]])
```

```
A.T
```

```
Array([[0, 2, 4],
       [1, 3, 5]], dtype=int32)
```

```
tf.transpose(A)
```

```
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[0, 2, 4],
       [1, 3, 5]], dtype=int32)>
```

対称行列は、正方行列のうち自分自身の転置に等しいものです: $\mathbf{A} = \mathbf{A}^\top$. 次の行列は対称である。

```
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T
```

```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

```
A = np.array([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T
```

```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

```
A = jnp.array([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T
```

```
Array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)
```

```
A = tf.constant([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == tf.transpose(A)
```

```
<tf.Tensor: shape=(3, 3), dtype=bool, numpy=
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])>
```

行列はデータセットを表現するのに便利である。通常、行は個々の記録に対応し、列は異なる属性に対応する。

5.3.4 テンソル

スカラー、ベクトル、行列だけでも機械学習の道のりをかなり進めるが、やがてはより高階のテンソルを扱う必要が出てきる。テンソルは n^{th}-order 配列への拡張を一般的に記述する方法を与えてくれる。テンソルクラスのソフトウェア

オブジェクトを「テンソル」と呼ぶのは、それらも任意の数の軸を持てるからである。数学的対象としての テンソルと、コード上での実装を同じ語で呼ぶのは紛らわしいかもしれないが、通常は文脈から意味が明らかである。一般のテンソルは特別な書体の大文字（たとえば X、Y、Z）で表し、そのインデックス付けの仕組み（たとえば x_{ijk} や $[\mathsf{X}]_{1,2i-1,3}$）は行列のそれから自然に拡張される。

画像を扱い始めると、テンソルはさらに重要になる。各画像は、高さ、幅、チャネルに対応する軸を持つ 3^{rd}-order テンソルとして与えられる。各空間位置では、各色（赤、緑、青）の強度がチャネル方向に並べられる。さらに、画像の集合はコード上では 4^{th}-order テンソルとして表され、個々の画像は第 1 軸に沿ってインデックス付けされる。高階テンソルは、ベクトルや行列と同様に、shape の成分数を増やすことで構成される。

```
torch.arange(24).reshape(2, 3, 4)
```

```
tensor([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]])
```

```
np.arange(24).reshape(2, 3, 4)
```

```
array([[[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]],

       [[12., 13., 14., 15.],
        [16., 17., 18., 19.],
        [20., 21., 22., 23.]]])
```

```
jnp.arange(24).reshape(2, 3, 4)
```

```
Array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]], dtype=int32)
```

```
tf.reshape(tf.range(24), (2, 3, 4))
```

```
<tf.Tensor: shape=(2, 3, 4), dtype=int32, numpy=
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]], dtype=int32)>
```

5.3.5 テンソル演算の基本的性質

スカラー、ベクトル、行列、および高階テンソルには、便利な性質がいくつかある。たとえば、要素ごとの演算は、入力と同じ shape を持つ出力を生成する。

```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone()  # Assign a copy of A to B by allocating new memory
A, A + B
```

```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  2.,  4.],
         [ 6.,  8., 10.]]))
```

```
A = np.arange(6).reshape(2, 3)
B = A.copy()  # Assign a copy of A to B by allocating new memory
A, A + B
```

```
(array([[0., 1., 2.],
        [3., 4., 5.]]),
 array([[ 0.,  2.,  4.],
        [ 6.,  8., 10.]]))
```

```
A = jnp.arange(6, dtype=jnp.float32).reshape(2, 3)
B = A
A, A + B
```

```
(Array([[0., 1., 2.],
        [3., 4., 5.]], dtype=float32),
 Array([[ 0.,  2.,  4.],
        [ 6.,  8., 10.]], dtype=float32))
```

```
A = tf.reshape(tf.range(6, dtype=tf.float32), (2, 3))
B = A  # No cloning of A to B by allocating new memory
A, A + B
```

```
(<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
 array([[0., 1., 2.],
        [3., 4., 5.]], dtype=float32)>,
 <tf.Tensor: shape=(2, 3), dtype=float32, numpy=
 array([[ 0.,  2.,  4.],
        [ 6.,  8., 10.]], dtype=float32)>)
```

2 つの行列の要素ごとの積は *Hadamard* 積と呼ばれます（$\odot$ で表す）。2 つの行列 $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ の Hadamard 積の各要素は次のように書ける。

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \tag{5.3.4}$$

```
A * B
```

```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

スカラーとテンソルの加算や乗算は、元のテンソルと同じ shape の結果を生成する。ここでは、テンソルの各要素にスカラーを加算（または乗算）している。

```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(tensor([[[ 2,  3,  4,  5],
          [ 6,  7,  8,  9],
          [10, 11, 12, 13]],

         [[14, 15, 16, 17],
          [18, 19, 20, 21],
          [22, 23, 24, 25]]]),
 torch.Size([2, 3, 4]))
```

```
a = 2
X = np.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(array([[[ 2.,  3.,  4.,  5.],
         [ 6.,  7.,  8.,  9.],
         [10., 11., 12., 13.]],

        [[14., 15., 16., 17.],
```

(continues on next page)

(continued from previous page)

```
        [18., 19., 20., 21.],
        [22., 23., 24., 25.]]]),
 (2, 3, 4))
```

```
a = 2
X = jnp.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(Array([[[ 2,  3,  4,  5],
         [ 6,  7,  8,  9],
         [10, 11, 12, 13]],

        [[14, 15, 16, 17],
         [18, 19, 20, 21],
         [22, 23, 24, 25]]], dtype=int32),
 (2, 3, 4))
```

```
a = 2
X = tf.reshape(tf.range(24), (2, 3, 4))
a + X, (a * X).shape
```

```
(<tf.Tensor: shape=(2, 3, 4), dtype=int32, numpy=
 array([[[ 2,  3,  4,  5],
         [ 6,  7,  8,  9],
         [10, 11, 12, 13]],

        [[14, 15, 16, 17],
         [18, 19, 20, 21],
         [22, 23, 24, 25]]], dtype=int32)>,
 TensorShape([2, 3, 4]))
```

5.3.6 リダクション

しばしば、テンソルの要素の総和を計算したいことがある。長さ n のベクトル $\mathbf{x}$ の要素の和を表すには、$\sum_{i=1}^{n} x_i$ と書きる。これには簡単な関数がある。

```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

```
(tensor([0., 1., 2.]), tensor(3.))
```

```
x = np.arange(3)
x, x.sum()
```

```
(array([0., 1., 2.]), array(3.))
```

```
x = jnp.arange(3, dtype=jnp.float32)
x, x.sum()
```

```
(Array([0., 1., 2.], dtype=float32), Array(3., dtype=float32))
```

```
x = tf.range(3, dtype=tf.float32)
x, tf.reduce_sum(x)
```

```
(<tf.Tensor: shape=(3,), dtype=float32, numpy=array([0., 1., 2.],␣
↪dtype=float32)>,
 <tf.Tensor: shape=(), dtype=float32, numpy=3.0>)
```

任意の shape のテンソルの要素の和を表すには、すべての軸にわたって単純に和を取ればよいである。たとえば、$m \times n$ 行列 $\mathbf{A}$ の要素の和は $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$ と書ける。

```
A.shape, A.sum()
```

```
(torch.Size([2, 3]), tensor(15.))
```

```
A.shape, A.sum()
```

```
((2, 3), array(15.))
```

```
A.shape, A.sum()
```

```
((2, 3), Array(15., dtype=float32))
```

```
A.shape, tf.reduce_sum(A)
```

```
(TensorShape([2, 3]), <tf.Tensor: shape=(), dtype=float32, numpy=15.0>)
```

デフォルトでは、sum 関数を呼ぶとテンソルはすべての軸に沿って リダクションされ、最終的にスカラーが得られる。ライブラリでは、テンソルをどの軸に沿ってリダクションするかを指定することもできる。行（軸 0）に沿ってすべての要素を足し合わせるには、sum で `axis=0` を指定する。入力行列は軸 0 に沿ってリダクションされて出力ベクトルを生成するため、この軸は出力の shape から消える。

```
A.shape, A.sum(axis=0).shape
```

```
(torch.Size([2, 3]), torch.Size([3]))
```

```
A.shape, A.sum(axis=0).shape
```

```
((2, 3), (3,))
```

```
A.shape, A.sum(axis=0).shape
```

```
((2, 3), (3,))
```

```
A.shape, tf.reduce_sum(A, axis=0).shape
```

```
(TensorShape([2, 3]), TensorShape([3]))
```

axis=1 を指定すると、列方向（軸 1）が、すべての列の要素を足し合わせることでリダクションされる。

```
A.shape, A.sum(axis=1).shape
```

```
(torch.Size([2, 3]), torch.Size([2]))
```

```
A.shape, A.sum(axis=1).shape
```

```
((2, 3), (2,))
```

```
A.shape, A.sum(axis=1).shape
```

```
((2, 3), (2,))
```

```
A.shape, tf.reduce_sum(A, axis=1).shape
```

```
(TensorShape([2, 3]), TensorShape([2]))
```

行と列の両方に沿って和を取って行列をリダクションすることは、行列のすべての要素を足し合わせることと同じである。

```
A.sum(axis=[0, 1]) == A.sum()  # Same as A.sum()
```

```
tensor(True)
```

```
A.sum(axis=[0, 1]) == A.sum()  # Same as A.sum()
```

```
array(True)
```

```
A.sum(axis=[0, 1]) == A.sum()  # Same as A.sum()
```

```
Array(True, dtype=bool)
```

```
tf.reduce_sum(A, axis=[0, 1]), tf.reduce_sum(A)  # Same as tf.reduce_sum(A)
```

```
(<tf.Tensor: shape=(), dtype=float32, numpy=15.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=15.0>)
```

関連する量として 平均、別名 アベレージがある。平均は、和を要素数の総数で割ることで求める。平均の計算は非常に頻繁に行われるため、sum と同様に使える専用のライブラリ関数がある。

```
A.mean(), A.sum() / A.numel()
```

```
(tensor(2.5000), tensor(2.5000))
```

```
A.mean(), A.sum() / A.size
```

```
(array(2.5), array(2.5))
```

```
A.mean(), A.sum() / A.size
```

```
(Array(2.5, dtype=float32), Array(2.5, dtype=float32))
```

```
tf.reduce_mean(A), tf.reduce_sum(A) / tf.size(A).numpy()
```

```
(<tf.Tensor: shape=(), dtype=float32, numpy=2.5>,
 <tf.Tensor: shape=(), dtype=float32, numpy=2.5>)
```

同様に、平均を計算する関数も特定の軸に沿ってテンソルをリダクションできる。

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(array([1.5, 2.5, 3.5]), array([1.5, 2.5, 3.5]))
```

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(Array([1.5, 2.5, 3.5], dtype=float32), Array([1.5, 2.5, 3.5], dtype=float32))
```

```
tf.reduce_mean(A, axis=0), tf.reduce_sum(A, axis=0) / A.shape[0]
```

```
(<tf.Tensor: shape=(3,), dtype=float32, numpy=array([1.5, 2.5, 3.5],␣
↪dtype=float32)>,
 <tf.Tensor: shape=(3,), dtype=float32, numpy=array([1.5, 2.5, 3.5],␣
↪dtype=float32)>)
```

5.3.7 非リダクション和

和や平均を計算する関数を呼ぶときに、軸の数を変えずに保つと便利な場合がある。これは、ブロードキャスト機構を使いたいときに重要である。

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

```
(array([[ 3.],
        [12.]]),
 (2, 1))
```

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

```
(Array([[ 3.],
        [12.]], dtype=float32),
 (2, 1))
```

```
sum_A = tf.reduce_sum(A, axis=1, keepdims=True)
sum_A, sum_A.shape
```

```
(<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
 array([[ 3.],
        [12.]], dtype=float32)>,
 TensorShape([2, 1]))
```

たとえば、sum_A は各行を足し合わせた後も 2 つの軸を保つので、ブロードキャストを使って A を sum_A で割ることができ、各行の和が 1 になる行列を作れる。

```
A / sum_A
```

```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

A の要素の累積和をある軸に沿って、たとえば axis=0（行ごと）で計算したければ、cumsum 関数を呼べる。設計上、この関数は入力テンソルをどの軸に沿ってもリダクションしない。

```
A.cumsum(axis=0)
```

```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

```
A.cumsum(axis=0)
```

```
array([[0., 1., 2.],
       [3., 5., 7.]])
```

```
A.cumsum(axis=0)
```

```
Array([[0., 1., 2.],
       [3., 5., 7.]], dtype=float32)
```

```
tf.cumsum(A, axis=0)
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0., 1., 2.],
       [3., 5., 7.]], dtype=float32)>
```

5.3.8 ドット積

ここまでで、要素ごとの演算、和、平均だけを扱ってきた。もしこれしかできないなら、線形代数が独立した節を持つ価値はない。幸い、ここからが面白くなる。最も基本的な演算の 1 つがドット積である。2 つのベクトル $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ に対して、その ドット積 $\mathbf{x}^\top \mathbf{y}$（内積、$\langle \mathbf{x}, \mathbf{y} \rangle$ とも呼ばれます）は、同じ位置にある要素の積の和です: $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$。

```
y = torch.ones(3, dtype = torch.float32)
x, y, torch.dot(x, y)
```

```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))
```

```
y = np.ones(3)
x, y, np.dot(x, y)
```

```
(array([0., 1., 2.]), array([1., 1., 1.]), array(3.))
```

```
y = jnp.ones(3, dtype = jnp.float32)
x, y, jnp.dot(x, y)
```

```
(Array([0., 1., 2.], dtype=float32),
 Array([1., 1., 1.], dtype=float32),
 Array(3., dtype=float32))
```

```
y = tf.ones(3, dtype=tf.float32)
x, y, tf.tensordot(x, y, axes=1)
```

```
(<tf.Tensor: shape=(3,), dtype=float32, numpy=array([0., 1., 2.],␣
↪dtype=float32)>,
 <tf.Tensor: shape=(3,), dtype=float32, numpy=array([1., 1., 1.],␣
↪dtype=float32)>,
 <tf.Tensor: shape=(), dtype=float32, numpy=3.0>)
```

同値な見方として、2 つのベクトルのドット積は、要素ごとの乗算の後に和を取ることで計算できる:

```
torch.sum(x * y)
```

```
tensor(3.)
```

```
np.sum(x * y)
```

```
array(3.)
```

```
jnp.sum(x * y)
```

```
Array(3., dtype=float32)
```

```
tf.reduce_sum(x * y)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=3.0>
```

ドット積は幅広い文脈で有用である。たとえば、ある値の集合をベクトル $\mathbf{x} \in \mathbb{R}^n$ で表し、重みの集合を $\mathbf{w} \in \mathbb{R}^n$ で表すと、重み $\mathbf{w}$ に従った $\mathbf{x}$ の値の重み付き和はドット積 $\mathbf{x}^\top \mathbf{w}$ として表せる。重みが非負で、かつ 1 に和が等しい、すなわち $\left(\sum_{i=1}^{n} w_i = 1\right)$ とき、ドット積は 重み付き平均を表す。2 つのベクトルを単位長に正規化すると、ドット積はそれらのなす角の余弦を表す。この節の後半で、この 長さの概念を正式に導入する。

5.3.9 行列–ベクトル積

ドット積の計算方法がわかったので、$m \times n$ 行列 $\mathbf{A}$ と n 次元ベクトル $\mathbf{x}$ の間の 積を理解し始められる。まず、行ベクトルの観点から行列を可視化する。

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \tag{5.3.5}$$

ここで各 $\mathbf{a}_i^\top \in \mathbb{R}^n$ は、行列 $\mathbf{A}$ の i^{th} 行を表す行ベクトルである。

行列–ベクトル積 $\mathbf{Ax}$ は、長さ m の列ベクトルにすぎず、その i^{th} 要素はドット積 $\mathbf{a}_i^\top \mathbf{x}$ です:

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}. \tag{5.3.6}$$

行列 $\mathbf{A} \in \mathbb{R}^{m \times n}$ による乗算は、ベクトルを $\mathbb{R}^n$ から $\mathbb{R}^m$ へ写す変換だと考えられる。このような変換は非常に有用である。たとえば、回転は特定の正方行列との乗算として表現できる。行列–ベクトル積は、前の層の出力からニューラルネットワークの各層の出力を計算する際の主要な計算も表している。

コードで行列–ベクトル積を表すには、`mv` 関数を使う。`A` の列方向の次元（軸 1 に沿った長さ）が `x` の次元（長さ）と同じでなければならないことに注意しよ。Python には便利な演算子 `@` があり、行列–ベクトル積と行列–行列積の両方を（引数に応じて）実行できる。したがって `A@x` と書ける。

```
A.shape, x.shape, torch.mv(A, x), A@x
```

```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]), tensor([ 5., 14.]))
```

```
A.shape, x.shape, np.dot(A, x)
```

```
((2, 3), (3,), array([ 5., 14.]))
```

```
A.shape, x.shape, jnp.matmul(A, x)
```

```
((2, 3), (3,), Array([ 5., 14.], dtype=float32))
```

```
A.shape, x.shape, tf.linalg.matvec(A, x)
```

```
(TensorShape([2, 3]),
 TensorShape([3]),
 <tf.Tensor: shape=(2,), dtype=float32, numpy=array([ 5., 14.], dtype=float32)>
↪)
```

5.3.10 行列–行列積

ドット積と行列–ベクトル積に慣れれば、行列–行列積は簡単に理解できる。

2 つの行列 $\mathbf{A} \in \mathbb{R}^{n \times k}$ と $\mathbf{B} \in \mathbb{R}^{k \times m}$ があるとする。

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}. \tag{5.3.7}$$

$\mathbf{A}$ の i^{th} 行を表す行ベクトルを $\mathbf{a}_i^\top \in \mathbb{R}^k$ とし、$\mathbf{B}$ の j^{th} 列を表す列ベクトルを

$\mathbf{b}_j \in \mathbb{R}^k$ とする。

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix}. \tag{5.3.8}$$

行列積 $\mathbf{C} \in \mathbb{R}^{n \times m}$ を作るには、各要素 c_{ij} を $\mathbf{A}$ の i^{th} 行と $\mathbf{B}$ の j^{th} 列のドット積、すなわち $\mathbf{a}_i^\top \mathbf{b}_j$ として計算するだけである。

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}. \tag{5.3.9}$$

行列–行列積 $\mathbf{AB}$ は、m 個の行列–ベクトル積または $m \times n$ 個のドット積を計算し、その結果をつなぎ合わせて $n \times m$ 行列を作るものだと考えられる。次のコード片では、`A` と `B` に対して行列積を行う。ここで `A` は 2 行 3 列の行列で、`B` は 3 行 4 列の行列である。乗算後、2 行 4 列の行列が得られる。

```
B = torch.ones(3, 4)
torch.mm(A, B), A@B
```

```
(tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]),
 tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]))
```

```
B = np.ones(shape=(3, 4))
np.dot(A, B)
```

```
array([[ 3.,  3.,  3.,  3.],
       [12., 12., 12., 12.]])
```

```
B = jnp.ones((3, 4))
jnp.matmul(A, B)
```

```
Array([[ 3.,  3.,  3.,  3.],
       [12., 12., 12., 12.]], dtype=float32)
```

```
B = tf.ones((3, 4), tf.float32)
tf.matmul(A, B)
```

```
<tf.Tensor: shape=(2, 4), dtype=float32, numpy=
array([[ 3.,  3.,  3.,  3.],
       [12., 12., 12., 12.]], dtype=float32)>
```

行列–行列積という用語は、しばしば単に 行列積と短縮され、Hadamard 積と混同してはならない。

5.3.11 ノルム

線形代数で最も有用な演算子のいくつかは ノルムである。直感的には、ベクトルのノルムはそれがどれだけ 大きいかを教えてくれる。たとえば、ℓ_2 ノルムはベクトルの（ユークリッド）長さを測りる。ここで私たちは、ベクトルの成分の大きさに関する サイズの概念（次元数ではない）を使っている。

ノルムは、ベクトルをスカラーに写す関数 $\|\cdot\|$ であり、次の 3 つの性質を満たする。

1. 任意のベクトル $\mathbf{x}$ について、ベクトル（のすべての要素）をスカラー $\alpha \in \mathbb{R}$ でスケールすると、そのノルムもそれに応じてスケールする:

$$\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|. \tag{5.3.10}$$

2. 任意のベクトル $\mathbf{x}$ と $\mathbf{y}$ について、ノルムは三角不等式を満たす:

$$\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|. \tag{5.3.11}$$

3. ベクトルのノルムは非負であり、ベクトルがゼロのときにのみ 0 になる:

$$\|\mathbf{x}\| > 0 \text{ for all } \mathbf{x} \neq 0. \tag{5.3.12}$$

多くの関数が有効なノルムであり、異なるノルムは異なるサイズの概念を表す。小学校の図形で直角三角形の斜辺を求めるときに学ぶユークリッドノルムは、ベクトルの要素の二乗和の平方根である。形式的には、これは ℓ_2 ノルムと呼ばれ、次のように表される。

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}. \tag{5.3.13}$$

`norm` メソッドは ℓ_2 ノルムを計算する。

```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

```
tensor(5.)
```

```
u = np.array([3, -4])
np.linalg.norm(u)
```

```
array(5.)
```

```
u = jnp.array([3.0, -4.0])
jnp.linalg.norm(u)
```

```
Array(5., dtype=float32)
```

```
u = tf.constant([3.0, -4.0])
tf.norm(u)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=5.0>
```

ℓ_1 ノルムもよく使われ、それに対応する尺度はマンハッタン距離と呼ばれる。定義により、ℓ_1 ノルムはベクトルの要素の絶対値の和である。

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| \,. \tag{5.3.14}$$

ℓ_2 ノルムと比べると、外れ値の影響を受けにくいである。ℓ_1 ノルムを計算するには、絶対値と和の演算を組み合わせる。

```
torch.abs(u).sum()
```

```
tensor(7.)
```

```
np.abs(u).sum()
```

```
array(7.)
```

```
jnp.linalg.norm(u, ord=1) # same as jnp.abs(u).sum()
```

```
Array(7., dtype=float32)
```

```
tf.reduce_sum(tf.abs(u))
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=7.0>
```

ℓ_2 ノルムと ℓ_1 ノルムはどちらも、より一般的な ℓ_p ノルムの特殊な場合である。

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}. \tag{5.3.15}$$

行列の場合は、事情がより複雑である。そもそも行列は、個々の要素の集合としても、ベクトルに作用して別のベクトルへ変換する対象としても見られる。たとえば、行列–ベクトル積 $\mathbf{Xv}$ が $\mathbf{v}$ に比べてどれだけ長くなりうるかを問うことができる。この考え方は、スペクトルノルムと呼ばれるものにつながりる。ここではまず、計算がずっと簡単な フロベニウスノルムを導入する。これは、行列の要素の二乗和の平方根として定義される。

$$\|\mathbf{X}\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}^2}. \tag{5.3.16}$$

フロベニウスノルムは、行列の形をしたベクトルに対する ℓ_2 ノルムのように振る舞いる。次の関数を呼ぶと、行列のフロベニウスノルムが計算される。

```
torch.norm(torch.ones((4, 9)))
```

```
tensor(6.)
```

```
np.linalg.norm(np.ones((4, 9)))
```

```
array(6.)
```

```
jnp.linalg.norm(jnp.ones((4, 9)))
```

```
Array(6., dtype=float32)
```

```
tf.norm(tf.ones((4, 9)))
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=6.0>
```

あまり先走りすぎたくはないが、これらの概念がなぜ有用なのかについての直感はすでに少し持っておける。深層学習では、しばしば最適化問題を解こうとする。観測データに割り当てられる確率を 最大化すること、推薦モデルに関連

する収益を 最大化すること、予測と正解観測値の間の距離を 最小化すること、同じ人物の写真の表現同士の距離を 最小化しつつ、異なる人物の写真の表現同士の距離を 最大化すること。これらの距離は深層学習アルゴリズムの目的関数を構成し、しばしばノルムとして表される。

5.3.12 議論

この節では、現代の深層学習のかなりの部分を理解するのに必要な線形代数をひととおり見てきた。とはいえ、線形代数にはまだまだ多くの内容があり、その多くは機械学習に有用である。たとえば、行列は因子に分解でき、その分解によって実世界のデータセットに潜む低次元構造を明らかにできることがある。機械学習には、行列分解とその高階テンソルへの一般化を用いてデータセットの構造を発見し、予測問題を解くことに焦点を当てたサブフィールドが丸ごと存在する。しかし、この本の焦点は深層学習である。そして、実際のデータセットに機械学習を適用して手を動かした後のほうが、より多くの数学を学ぶ意欲が高まると私たちは考えている。そのため、後でさらに数学を導入する余地は残しつつも、ここでこの節を締めくくりる。

もっと線形代数を学びたいなら、優れた書籍やオンライン資料がたくさんある。より発展的な速習コースとしては、Strang (1993), Kolter (2008), Petersen and Pedersen (2008) を参照されたい。

要点をまとめると:

- スカラー、ベクトル、行列、テンソルは線形代数で使われる基本的な数学的対象であり、それぞれ 0、1、2、および任意個の軸を持つ。
- テンソルは、インデックス付けや `sum`、`mean` などの演算によって、指定した軸に沿ってスライスしたりリダクションしたりできる。
- 要素ごとの積は Hadamard 積と呼ばれる。これに対して、ドット積、行列–ベクトル積、行列–行列積は要素ごとの演算ではなく、一般に入力とは異なる shape を持つ対象を返す。
- Hadamard 積と比べると、行列–行列積は計算にかなり時間がかかります（2 次時間ではなく 3 次時間）。
- ノルムはベクトル（または行列）の大きさに関するさまざまな概念を捉え、2 つのベクトルの差に適用して距離を測るのによく使われる。
- よく使われるベクトルノルムには ℓ_1 ノルムと ℓ_2 ノルムがあり、よく使われる行列ノルムには スペクトルノルムと フロベニウスノルムがある。

5.3.13 演習

1. 行列の転置の転置は元の行列そのものであることを証明せよ: $(\mathbf{A}^\top)^\top = \mathbf{A}$。
2. 2 つの行列 $\mathbf{A}$ と $\mathbf{B}$ について、和と転置が可換であることを示せ: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$。

3. 任意の正方行列 $\mathbf{A}$ について、$\mathbf{A} + \mathbf{A}^\top$ は常に対称か。前の 2 つの演習の結果だけを使って証明できるか。
4. この節では shape が (2, 3, 4) のテンソル `X` を定義した。`len(X)` の出力は何だろうか。コードを実行せずに答えを書き、その後コードで確認しよ。
5. 任意の shape のテンソル `X` について、`len(X)` は常に `X` のある軸の長さに対応するか。その軸はどれですか。
6. `A / A.sum(axis=1)` を実行して何が起こるか見よ。結果を分析できるか。
7. マンハッタンの中心部で 2 点間を移動するとき、座標、すなわち通りと街路の観点で、どれだけの距離を移動する必要があるか。斜めに移動できるか。
8. shape が (2, 3, 4) のテンソルを考える。軸 0、1、2 に沿った和の出力の shape はそれぞれ何ですか。
9. 3 つ以上の軸を持つテンソルを `linalg.norm` 関数に入力して、その出力を観察しよ。この関数は任意の shape のテンソルに対して何を計算するか。
10. たとえば $\mathbf{A} \in \mathbb{R}^{2^{10} \times 2^{16}}$, $\mathbf{B} \in \mathbb{R}^{2^{16} \times 2^5}$, $\mathbf{C} \in \mathbb{R}^{2^5 \times 2^{14}}$ のような 3 つの大きな行列を、ガウス乱数で初期化したとする。積 $\mathbf{ABC}$ を計算したいとき、$(\mathbf{AB})\mathbf{C}$ と $\mathbf{A}(\mathbf{BC})$ のどちらで計算するかによって、メモリ使用量や速度に違いはあるか。なぜですか。
11. たとえば $\mathbf{A} \in \mathbb{R}^{2^{10} \times 2^{16}}$, $\mathbf{B} \in \mathbb{R}^{2^{16} \times 2^5}$, $\mathbf{C} \in \mathbb{R}^{2^5 \times 2^{16}}$ のような 3 つの大きな行列を考える。$\mathbf{AB}$ と $\mathbf{AC}^\top$ のどちらを計算するかによって速度に違いはあるか。なぜですか。もし $\mathbf{C} = \mathbf{B}^\top$ をメモリを複製せずに初期化したら何が変わりますか。なぜですか。
12. たとえば $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{100 \times 200}$ の 3 つの行列を考える。$[\mathbf{A}, \mathbf{B}, \mathbf{C}]$ をスタックして 3 つの軸を持つテンソルを構成しよ。次元数はいくつですか。第 3 軸の第 2 成分を取り出して $\mathbf{B}$ を復元しよ。答えが正しいことを確認しよ。

5.4 微分積分

長い間、円の面積をどのように計算するかは謎のままでした。その後、古代ギリシャの数学者アルキメデスが、円の内側に頂点数を増やした多角形を次々と内接させるという画期的なアイデアを考案した (Fig. 5.4.1)。n 個の頂点をもつ多角形については、n 個の三角形が得られる。円をより細かく分割するにつれて、各三角形の高さは半径 r に近づきる。同時に、弧と割線の比が頂点数が十分大きいときに 1 に近づくので、底辺は $2\pi r/n$ に近づきる。したがって、多角形の面積は $n \cdot r \cdot \frac{1}{2}(2\pi r/n) = \pi r^2$ に近づきる。

この極限過程は、微分学と 積分学の両方の根底にある。前者は、引数を操作することで関数値をどのように増減させるかを教えてくれる。これは、損失関数を減らすためにパラメータを繰り返し更新する深層学習で直面する 最適化

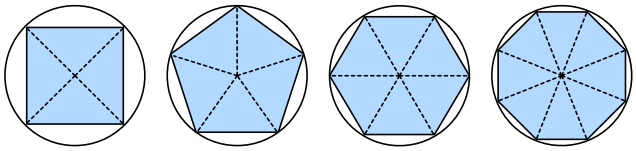


Fig. 5.4.1 極限過程として円の面積を求める。

問題において非常に役立ちる。最適化はモデルを訓練データに適合させるプロセスであり、微分積分学はそのための重要な基礎知識である。ただし、最終的な目標は これまで見たことのないデータで良い性能を発揮することだという点を忘れてはいけない。この問題は 汎化と呼ばれ、他の章で重要な焦点となる。

```
%matplotlib inline
from d2l import torch as d2l
from matplotlib_inline import backend_inline
import numpy as np
```

```
%matplotlib inline
from d2l import mxnet as d2l
from matplotlib_inline import backend_inline
from mxnet import np, npx
npx.set_np()
```

```
%matplotlib inline
from d2l import jax as d2l
from matplotlib_inline import backend_inline
import numpy as np
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
from matplotlib_inline import backend_inline
import numpy as np
```

5.4.1 導関数と微分

簡単に言えば、導関数とは、引数の変化に対する関数の変化率である。導関数は、各パラメータを微小な量だけ 増減させたときに、損失関数がどれだけ変化するかを示してくれる。形式的には、スカラーからスカラーへの写像である関数 $f : \mathbb{R} \to \mathbb{R}$ に対して、点 x における f の 導関数は次のように定義される

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{5.4.1}$$

右辺のこの項は 極限と呼ばれ、ある変数が特定の値に近づくときに式の値がどうなるかを教えてくれる。この極限は、摂動 h と関数値の変化 $f(x+h) - f(x)$ の比が、その大きさを 0 に近づけたときにどの値に収束するかを示している。

$f'(x)$ が存在するとき、f は x において 微分可能であるといいる。また、ある集合、たとえば区間 $[a, b]$ 上のすべての x で $f'(x)$ が存在するとき、f はその集合上で微分可能であるという。すべての関数が微分可能なわけではなく、最適化したい多くの関数、たとえば精度や受信者動作特性曲線下面積（AUC）もその例である。しかし、損失の導関数を計算することは深層ニューラルネットワークを訓練するほぼすべてのアルゴリズムにおいて極めて重要なステップであるため、しばしば微分可能な 代理を最適化する。

導関数 $f'(x)$ は、x に関する $f(x)$ の 瞬間的な変化率として解釈できる。例を通して直感を養いよう。$u = f(x) = 3x^2 - 4x$ と定義する。

```
def f(x):
    return 3 * x ** 2 - 4 * x
```

$x = 1$ とすると、$\frac{f(x+h)-f(x)}{h}$ が h が 0 に近づくにつれて 2 に近づくことがわかる。この実験は数学的証明ほど厳密ではないが、確かに $f'(1) = 2$ であることがすぐにわかる。

```
for h in 10.0**np.arange(-1, -6, -1):
    print(f'h={h:.5f}, numerical limit={(f(1+h)-f(1))/h:.5f}')
```

```
h=0.10000, numerical limit=2.30000
h=0.01000, numerical limit=2.03000
h=0.00100, numerical limit=2.00300
h=0.00010, numerical limit=2.00030
h=0.00001, numerical limit=2.00003
```

導関数には、いくつか同値な記法がある。$y = f(x)$ とすると、次の表現はすべて同値である。

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx}f(x) = Df(x) = D_x f(x), \tag{5.4.2}$$

ここで、$\frac{d}{dx}$ と D は 微分作用素である。以下に、いくつかの一般的な関数の導関数を示す。

$$\begin{aligned}
&\frac{d}{dx}C = 0 && \text{任意の定数 } C \text{ に対して} \\
&\frac{d}{dx}x^n = nx^{n-1} && n \neq 0 \text{ のとき} \\
&\frac{d}{dx}e^x = e^x \\
&\frac{d}{dx}\ln x = x^{-1}.
\end{aligned} \tag{5.4.3}$$

微分可能な関数から合成された関数も、しばしばそれ自体が微分可能である。次の規則は、任意の微分可能な関数 f と g、および定数 C の合成を扱う際に役立ちる。

$$\begin{aligned}
\frac{d}{dx}[Cf(x)] &= C\frac{d}{dx}f(x) && \text{定数倍の法則}\\
\frac{d}{dx}[f(x)+g(x)] &= \frac{d}{dx}f(x)+\frac{d}{dx}g(x) && \text{和の法則}\\
\frac{d}{dx}[f(x)g(x)] &= f(x)\frac{d}{dx}g(x)+g(x)\frac{d}{dx}f(x) && \text{積の法則}\\
\frac{d}{dx}\frac{f(x)}{g(x)} &= \frac{g(x)\frac{d}{dx}f(x)-f(x)\frac{d}{dx}g(x)}{g^2(x)} && \text{商の法則}
\end{aligned} \tag{5.4.4}$$

これを用いると、次のようにして $3x^2-4x$ の導関数を求められる。

$$\frac{d}{dx}[3x^2-4x] = 3\frac{d}{dx}x^2-4\frac{d}{dx}x = 6x-4. \tag{5.4.5}$$

$x=1$ を代入すると、確かにこの位置での導関数は 2 になる。導関数は、ある特定の位置における関数の 傾きを教えてくれることに注意しよ。

5.4.2 可視化ユーティリティ

matplotlib ライブラリを使って関数の傾きを可視化できる。いくつかの関数を定義する必要がある。名前が示すように、use_svg_display は matplotlib により鮮明な画像のために SVG 形式でグラフを出力するよう指示す。コメント #@save は特別な修飾子で、任意の関数、クラス、その他のコードブロックを d2l パッケージに保存し、後でコードを繰り返さずにたとえば d2l.use_svg_display() のように呼び出せるようにする。

```
def use_svg_display():  #@save
    """Jupyter でプロットを表示するために svg 形式を使う。"""
    backend_inline.set_matplotlib_formats('svg')
```

便利なことに、set_figsize で図のサイズを設定できる。from matplotlib import pyplot as plt という import 文は d2l パッケージ内で #@save によりマークされているので、d2l.plt を呼び出せる。

```
def set_figsize(figsize=(3.5, 2.5)):  #@save
    """matplotlib の図のサイズを設定する。"""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

set_axes 関数は、ラベル、範囲、スケールを含む属性を軸に関連付けることができる。

```
#@save
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """matplotlib の軸を設定する。"""
```

(continues on next page)

(continued from previous page)

```
    axes.set_xlabel(xlabel), axes.set_ylabel(ylabel)
    axes.set_xscale(xscale), axes.set_yscale(yscale)
    axes.set_xlim(xlim),     axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()
```

これら 3 つの関数を使うと、複数の曲線を重ねて描く plot 関数を定義できる。ここでのコードの多くは、入力のサイズと形状が一致することを確認しているだけである。

```
#@save
def plot(X, Y=None, xlabel=None, ylabel=None, legend=[], xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None):
    """ データ点をプロットする。"""

    def has_one_axis(X):  # X (tensor or list) が 1 軸なら True
        return (hasattr(X, "ndim") and X.ndim == 1 or isinstance(X, list)
                and not hasattr(X[0], "__len__"))

    if has_one_axis(X): X = [X]
    if Y is None:
        X, Y = [[]] * len(X), X
    elif has_one_axis(Y):
        Y = [Y]
    if len(X) != len(Y):
        X = X * len(Y)

    set_figsize(figsize)
    if axes is None:
        axes = d2l.plt.gca()
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        axes.plot(x,y,fmt) if len(x) else axes.plot(y,fmt)
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
```

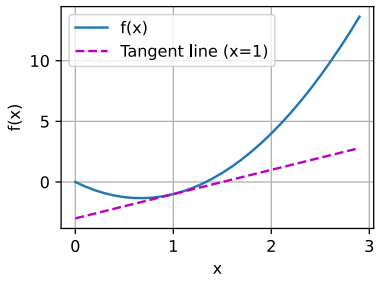
これで、関数 $u = f(x)$ と、$x = 1$ におけるその接線 $y = 2x - 3$ を描画できる。ここで係数 2 は接線の傾きである。

```
x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2 * x - 3], 'x', 'f(x)', legend=['f(x)', 'Tangent line (x=1)'])
```

5.4.3 偏導関数と勾配

ここまでは、1 変数だけの関数を微分してきた。深層学習では、多変数の関数も扱う必要がある。そのような 多変数関数に適用される導関数の概念を簡単に導入する。

$y = f(x_1, x_2, \ldots, x_n)$ を n 個の変数をもつ関数とする。i^{th} パラメータ x_i に関する y



の 偏導関数は

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{5.4.6}$$

$\frac{\partial y}{\partial x_i}$ を計算するには、$x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ を定数として扱い、x_i に関する y の導関数を計算すればよいである。偏導関数の記法には次のようなものがあり、どれも同じ意味である。

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = \partial_{x_i} f = \partial_i f = f_{x_i} = f_i = D_i f = D_{x_i} f. \tag{5.4.7}$$

多変数関数のすべての変数に関する偏導関数を連結すると、関数の 勾配と呼ばれるベクトルが得られる。関数 $f : \mathbb{R}^n \to \mathbb{R}$ の入力が n 次元ベクトル $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$ で、出力がスカラーであるとする。$\mathbf{x}$ に関する関数 f の勾配は、n 個の偏導関数からなるベクトルである。

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\partial_{x_1} f(\mathbf{x}), \partial_{x_2} f(\mathbf{x}), \ldots \partial_{x_n} f(\mathbf{x})\right]^\top. \tag{5.4.8}$$

曖昧さがなければ、$\nabla_{\mathbf{x}} f(\mathbf{x})$ は通常 $\nabla f(\mathbf{x})$ と書かれる。多変数関数を微分する際には、次の規則が役立ちる。

- すべての $\mathbf{A} \in \mathbb{R}^{m \times n}$ について、$\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$ および $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$ が成り立ちる。
- 正方行列 $\mathbf{A} \in \mathbb{R}^{n \times n}$ については、$\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$ が成り立ち、特に $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$ である。

同様に、任意の行列 $\mathbf{X}$ について、$\nabla_{\mathbf{X}} \|\mathbf{X}\|_{\mathrm{F}}^2 = 2\mathbf{X}$ が成り立ちる。

5.4.4 連鎖律

深層学習では、扱う勾配はしばしば計算が難しくなる。なぜなら、私たちは深く入れ子になった関数（関数の中の関数の中の関数…）を扱っているからである。幸いなことに、連鎖律がこれを処理してくれる。1 変数関数に戻ると、$y = f(g(x))$ であり、基礎となる関数 $y = f(u)$ と $u = g(x)$ がどちらも微分可能だとする。連鎖律は次のように述べる。

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}. \tag{5.4.9}$$

多変数関数に戻ると、$y = f(\mathbf{u})$ が変数 $u_1, u_2, \ldots, u_m$ をもち、各 $u_i = g_i(\mathbf{x})$ が変数 $x_1, x_2, \ldots, x_n$ をもつ、すなわち $\mathbf{u} = g(\mathbf{x})$ とする。このとき連鎖律は次のように述

べる。

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial u_1}\frac{\partial u_1}{\partial x_i} + \frac{\partial y}{\partial u_2}\frac{\partial u_2}{\partial x_i} + \ldots + \frac{\partial y}{\partial u_m}\frac{\partial u_m}{\partial x_i} \text{ したがって } \nabla_{\mathbf{x}} y = \mathbf{A}\nabla_{\mathbf{u}} y, \quad (5.4.10)$$

ここで $\mathbf{A} \in \mathbb{R}^{n\times m}$ は、ベクトル $\mathbf{u}$ のベクトル $\mathbf{x}$ に関する導関数を含む 行列である。したがって、勾配を評価するにはベクトルと行列の積を計算する必要がある。これが、線形代数が深層学習システムを構築するうえで非常に重要な構成要素である主要な理由の一つである。

5.4.5 議論

ここでは深い話題の表面を少しなぞったにすぎないが、すでにいくつかの概念が見えてきる。第一に、微分の合成規則は日常的に適用できるため、勾配を 自動的に計算できる。この作業には創造性が不要なので、私たちは他のことに認知資源を集中できる。第二に、ベクトル値関数の導関数を計算するには、出力から入力へと変数の依存グラフをたどりながら行列を掛け合わせる必要がある。特に、このグラフは関数を評価するときには 順方向に、勾配を計算するときには 逆方向にたどられる。後の章では、連鎖律を適用する計算手順である逆伝播を正式に導入する。

最適化の観点から見ると、勾配によって損失を下げるためにモデルのパラメータをどのように動かすべきかを決定できる。そして、この本全体で使う最適化アルゴリズムの各ステップでは、勾配の計算が必要になる。

5.4.6 演習

1. ここまでは導関数の規則を当然のものとして扱ってきた。定義と極限を用いて、(i) $f(x) = c$, (ii) $f(x) = x^n$, (iii) $f(x) = e^x$, (iv) $f(x) = \log x$ の性質を証明しよ。
2. 同様に、積の法則、和の法則、商の法則を第一原理から証明しよ。
3. 定数倍の法則が積の法則の特殊な場合として導かれることを証明しよ。
4. $f(x) = x^x$ の導関数を求めよ。
5. ある x に対して $f'(x) = 0$ であるとはどういう意味だろうか。そのようなことが成り立つ関数 f と位置 x の例を挙げよ。
6. 関数 $y = f(x) = x^3 - \frac{1}{x}$ を描画し、$x = 1$ における接線も描画しよ。
7. 関数 $f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$ の勾配を求めよ。
8. 関数 $f(\mathbf{x}) = \|\mathbf{x}\|_2$ の勾配は何だろうか。$\mathbf{x} = \mathbf{0}$ のときはどうなるか？
9. $u = f(x, y, z)$ かつ $x = x(a, b)$, $y = y(a, b)$, $z = z(a, b)$ の場合の連鎖律を書けますか？
10. 逆写像をもつ関数 $f(x)$ が与えられたとき、その逆関数 $f^{-1}(x)$ の導関数を求めよ。ここで $f^{-1}(f(x)) = x$ かつ逆に $f(f^{-1}(y)) = y$ である。ヒント: 導出ではこれらの性質を使っよ。

5.5 自動微分

Section 5.4 で思い出したように、導関数を計算することは、深層ネットワークを学習させるために用いるすべての最適化アルゴリズムにおいて極めて重要なステップである。計算自体は単純であるが、手作業で求めるのは面倒で誤りも起こりやすく、モデルが複雑になるほどこうした問題はさらに大きくなる。

幸いなことに、現代のあらゆる深層学習フレームワークは、自動微分（しばしば *autograd* と略される）を提供し、この手間のかかる作業を自動化してくれる。データを各関数へ順に通していくと、フレームワークは、各値が他の値にどのように依存しているかを追跡する 計算グラフを構築する。導関数を計算するには、自動微分はこのグラフを逆向きにたどり、連鎖律を適用する。このように連鎖律を適用する計算アルゴリズムは 逆伝播と呼ばれる。

autograd ライブラリはここ 10 年ほどで大きな注目を集めてきたが、その歴史は長く、最初期の autograd に関する言及は半世紀以上前にさかのぼります (Wengert, 1964)。現代の逆伝播の核となる考え方は 1980 年の博士論文にまでさかのぼり (Speelpenning, 1980)、1980 年代後半にさらに発展した (Griewank, 1989)。逆伝播は勾配を計算するための標準的な方法になっているが、唯一の選択肢ではない。たとえば、Julia プログラミング言語では順伝播が用いられている (Revels *et al.*, 2016)。方法を探る前に、まずは autograd パッケージを使いこなしよう。

```
import torch
```

```
from mxnet import autograd, np, npx
npx.set_np()
```

```
from jax import numpy as jnp
```

```
import tensorflow as tf
```

5.5.1 単純な関数

関数 $y = 2\mathbf{x}^\top \mathbf{x}$ を、列ベクトル $\mathbf{x}$ に関して微分したいとしよう。まず、x に初期値を割り当てる。

```
x = torch.arange(4.0)
x
```

```
tensor([0., 1., 2., 3.])
```

```
x = np.arange(4.0)
x
```

```
[07:04:30] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
array([0., 1., 2., 3.])
```

```
x = jnp.arange(4.0)
x
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
Array([0., 1., 2., 3.], dtype=float32)
```

```
x = tf.range(4, dtype=tf.float32)
x
```

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([0., 1., 2., 3.],␣
↪dtype=float32)>
```

y を $\mathbf{x}$ に関して微分する前に、その勾配を保存する場所が必要である。一般に、導関数を求めるたびに新しいメモリを割り当てることは避ける。というのも、深層学習では同じパラメータに関する導関数を何度も連続して計算する必要があり、メモリ不足に陥る危険があるからである。スカラー値関数のベクトル $\mathbf{x}$ に関する勾配は、$\mathbf{x}$ と同じ形状をもつベクトル値になる。

```
# x = torch.arange(4.0, requires_grad=True) としてもよい
x.requires_grad_(True)
x.grad  # 勾配の初期値はデフォルトで None
```

```
# `attach_grad` を呼び出して、テンソルの勾配用メモリを確保する
x.attach_grad()
# `x` に関する勾配を計算した後は、`grad` 属性を通じて
# その値にアクセスできる。初期値は 0 である
x.grad
```

```
array([0., 0., 0., 0.])
```

```
y = lambda x: 2 * jnp.dot(x, x)
y(x)
```

```
Array(28., dtype=float32)
```

```
x = tf.Variable(x)
```

次に、x の関数を計算して、その結果を y に代入する。

```
y = 2 * torch.dot(x, x)
y
```

```
tensor(28., grad_fn=<MulBackward0>)
```

```
# 計算グラフを構築するため、コードは `autograd.record` スコープ内にある
with autograd.record():
    y = 2 * np.dot(x, x)
y
```

```
array(28.)
```

```
from jax import grad
# `grad` 変換は、元の関数の勾配を計算する Python 関数を返す
x_grad = grad(y)(x)
x_grad
```

```
Array([ 0.,  4.,  8., 12.], dtype=float32)
```

```
# すべての計算をテープに記録する
with tf.GradientTape() as t:
    y = 2 * tf.tensordot(x, x, axes=1)
y
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=28.0>
```

これで y を x に関して微分できる。backward メソッドを呼び出す。次に、x の grad 属性を通じて勾配にアクセスできる。

```
y.backward()
x.grad
```

```
tensor([ 0.,  4.,  8., 12.])
```

```
y.backward()
x.grad
```

```
[07:04:30] ../src/base.cc:48: GPU context requested, but no GPUs found.
```

```
array([ 0.,  4.,  8., 12.])
```

```
x_grad == 4 * x
```

```
Array([ True,  True,  True,  True], dtype=bool)
```

```
x_grad = t.gradient(y, x)
x_grad
```

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([ 0.,  4.,  8., 12.],␣
↪dtype=float32)>
```

関数 $y = 2\mathbf{x}^\top\mathbf{x}$ の $\mathbf{x}$ に関する勾配は $4\mathbf{x}$ になることはすでに分かっている。これで、自動的に計算された勾配と期待される結果が一致することを確認できる。

```
x.grad == 4 * x
```

```
tensor([True, True, True, True])
```

```
x.grad == 4 * x
```

```
array([ True,  True,  True,  True])
```

```
y = lambda x: x.sum()
grad(y)(x)
```

```
Array([1., 1., 1., 1.], dtype=float32)
```

```
x_grad == 4 * x
```

```
<tf.Tensor: shape=(4,), dtype=bool, numpy=array([ True,  True,  True,  True])>
```

では、x の別の関数を計算して、その勾配を求めよう。PyTorch では、新しい勾配を記録しても勾配バッファは自動的にはリセットされない。その代わり、新しい勾配はすでに保存されている勾配に加算される。この挙動は、複数の目的関数の和を最適化したいときに便利である。勾配バッファをリセットするには、次のように x.grad.zero_() を呼び出す。

```
x.grad.zero_()  # 勾配をリセットする
y = x.sum()
y.backward()
x.grad
```

```
tensor([1., 1., 1., 1.])
```

```
with autograd.record():
    y = x.sum()
y.backward()
x.grad  # 新しく計算された勾配で上書きされる
```

```
array([1., 1., 1., 1.])
```

```
y = lambda x: x * x
# grad はスカラー出力関数に対してのみ定義される
grad(lambda x: y(x).sum())(x)
```

```
Array([0., 2., 4., 6.], dtype=float32)
```

```
with tf.GradientTape() as t:
    y = tf.reduce_sum(x)
t.gradient(y, x)  # 新しく計算された勾配で上書きされる
```

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([1., 1., 1., 1.],␣
↪dtype=float32)>
```

5.5.2 スカラーでない変数の逆伝播

y がベクトルのとき、y の x に関する導関数を表す最も自然な表現は、ヤコビアンと呼ばれる行列である。これは、y の各成分について x の各成分に関する偏導関数を含む。同様に、y と x がより高次元であれば、微分の結果はさらに高次のテンソルになることもある。

ヤコビアンはいくつかの高度な機械学習技法では現れるが、より一般的には、y の各成分の勾配を x の成分ごとに合計して、x と同じ形状のベクトルを得たいことが多いである。たとえば、訓練例の バッチごとに別々に計算された損失関数の値を表すベクトルを扱うことがよくある。この場合、私たちが欲しいのは各例ごとに個別に計算された勾配を足し合わせることだけである。

深層学習フレームワークはスカラーでないテンソルの勾配の解釈がそれぞれ異なるため、PyTorch は混乱を避けるための手順をいくつか用意している。スカラーでない対象に対して backward を呼び出すと、それをスカラーに縮約する方法を PyTorch に伝えない限りエラーになる。より形式的には、backward が $\partial_{\mathbf{x}}\mathbf{y}$ ではなく $\mathbf{v}^\top\partial_{\mathbf{x}}\mathbf{y}$ を計算するようにするためのベクトル $\mathbf{v}$ を与える必要がある。この先は少し分かりにくいかもしれないが、後で明らかになる理由により、この引数（$\mathbf{v}$ を表すもの）は gradient と名付けられている。より詳しい説明は、Yang Zhang の Medium の記事[44] を参照しよ。



```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))  # より高速: y.sum().backward()
x.grad
```

```
tensor([0., 2., 4., 6.])
```

```
with autograd.record():
    y = x * x
y.backward()
x.grad  # y = sum(x * x) の勾配に等しい
```

```
array([0., 2., 4., 6.])
```

```
import jax

y = lambda x: x * x
# jax.lax のプリミティブは XLA 演算の Python ラッパーである
u = jax.lax.stop_gradient(y(x))
z = lambda x: u * x

grad(lambda x: z(x).sum())(x) == y(x)
```

```
Array([ True,  True,  True,  True], dtype=bool)
```

```
with tf.GradientTape() as t:
    y = x * x
t.gradient(y, x)  # y = tf.reduce_sum(x * x) と同じ
```

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([0., 2., 4., 6.],␣
↪dtype=float32)>
```

5.5.3 計算の切り離し

ときには、いくつかの計算を記録された計算グラフの外に出したいことがある。たとえば、入力を使って補助的な中間項を作るが、その項については勾配を計算したくないとする。この場合、最終結果からそれぞれの計算グラフを 切り離す必要がある。次の簡単な例でこれを明確にしよう。z = x * y かつ y = x * x であるが、y を介して伝わる影響ではなく、z に対する x の 直接的な影響に注目したいとする。この場合、y と同じ値を持つ新しい変数 u を作れるが、その来歴（どのように生成されたか）は消去される。したがって u はグラフ内に祖先を持たず、勾配は u を通って x へ流れない。たとえば、z = x * u の勾配を取ると、結果は u になる（z = x * x * x だから 3 * x * x になると予想したかもしれないが、そうはならない）。

```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

z.sum().backward()
x.grad == u
```

```
tensor([True, True, True, True])
```

```
with autograd.record():
    y = x * x
    u = y.detach()
    z = u * x
z.backward()
x.grad == u
```

```
array([ True,  True,  True,  True])
```

```
grad(lambda x: y(x).sum())(x) == 2 * x
```

```
Array([ True,  True,  True,  True], dtype=bool)
```

```
# 計算グラフを保持するために persistent=True を設定する。
# これにより t.gradient を複数回実行できる
with tf.GradientTape(persistent=True) as t:
    y = x * x
    u = tf.stop_gradient(y)
    z = u * x

x_grad = t.gradient(z, x)
x_grad == u
```

```
<tf.Tensor: shape=(4,), dtype=bool, numpy=array([ True,  True,  True,  True])>
```

この手順は z へ至るグラフから y の祖先を切り離すが、y へ至る計算グラフ自体は残っているので、x に関する y の勾配を計算できる。

```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

```
tensor([True, True, True, True])
```

```
y.backward()
x.grad == 2 * x
```

```
array([ True,  True,  True,  True])
```

```
def f(a):
    b = a * 2
    while jnp.linalg.norm(b) < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

```
t.gradient(y, x) == 2 * x
```

```
<tf.Tensor: shape=(4,), dtype=bool, numpy=array([ True,  True,  True,  True])>
```

5.5.4 勾配と Python の制御フロー

これまで、入力から出力までの経路が z = x * x * x のような関数によって明確に定義されている場合を見てきた。プログラミングでは、結果の計算方法にもっと自由度がある。たとえば、補助変数に依存させたり、中間結果に応じて

条件分岐を行ったりできる。自動微分を使う利点の一つは、たとえ計算グラフを構築するのに Python の制御フローの迷路を通り抜ける必要があっても（たとえば条件分岐、ループ、任意の関数呼び出し）、最終的に得られる変数の勾配を計算できることである。これを示すために、次のコード片を考えよう。ここでは `while` ループの反復回数と `if` 文の評価の両方が入力 `a` の値に依存している。

```
def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

```
def f(a):
    b = a * 2
    while np.linalg.norm(b) < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

```
from jax import random
a = random.normal(random.PRNGKey(1), ())
d = f(a)
d_grad = grad(f)(a)
```

```
def f(a):
    b = a * 2
    while tf.norm(b) < 1000:
        b = b * 2
    if tf.reduce_sum(b) > 0:
        c = b
    else:
        c = 100 * b
    return c
```

以下では、この関数にランダムな値を入力として渡して呼び出す。入力は確率変数なので、計算グラフがどのような形になるかは分からない。しかし、特定の入力に対して `f(a)` を実行するたびに、特定の計算グラフが実現され、その後 `backward` を実行できる。

```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

```
a = np.random.normal()
a.attach_grad()
with autograd.record():
    d = f(a)
d.backward()
```

```
d_grad == d / a
```

```
Array(True, dtype=bool)
```

```
a = tf.Variable(tf.random.normal(shape=()))
with tf.GradientTape() as t:
    d = f(a)
d_grad = t.gradient(d, a)
d_grad
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=8192.0>
```

デモのために少し作為的な関数 f であるが、入力への依存は非常に単純である。それは、区分的に定義されたスケールをもつ a の 線形関数である。したがって、f(a) / a は定数成分からなるベクトルであり、さらに f(a) / a は a に関する f(a) の勾配と一致するはずである。

```
a.grad == d / a
```

```
tensor(True)
```

```
a.grad == d / a
```

```
array(True)
```

```
d_grad == d / a
```

```
<tf.Tensor: shape=(), dtype=bool, numpy=True>
```

動的な制御フローは深層学習で非常に一般的である。たとえば、テキストを処理するとき、計算グラフは入力の長さに依存する。このような場合、自動微分は統計モデリングにとって不可欠である。というのも、勾配を *a priori* に計算することは不可能だからである。

5.5.5 議論

これで、自動微分の力を少し体験できたはずである。導関数を自動的かつ効率的に計算するライブラリの発展は、深層学習の実践者にとって生産性を大きく高めるものであり、彼らが単純作業ではないことに集中できるようにしてくれた。さらに、autograd を使えば、紙と鉛筆での手計算では膨大な時間がかかりすぎるような巨大なモデルも設計できる。興味深いことに、私たちは autograd を使ってモデルを（統計的な意味で）最適化するが、autograd ライブラリ自体の（計算機科学的な意味での）最適化は、フレームワーク設計者にとって非常に重要な関心事である豊かな研究分野である。ここでは、コンパイラやグラフ操作の技術が用いられ、最も迅速でメモリ効率のよい方法で結果を計算する。

今のところ、次の基本を覚えておきよう。(i) 導関数を求めたい変数に勾配を付与する; (ii) 目的値の計算を記録する; (iii) 逆伝播関数を実行する; (iv) 得られた勾配にアクセスする。

5.5.6 演習

1. 2 階導関数を計算するのが 1 階導関数よりはるかに高コストなのはなぜですか。

2. 逆伝播の関数を実行した直後に、もう一度実行してみよ。何が起こるか調べよ。

3. d を a に関して微分する制御フローの例で、変数 a をランダムなベクトルや行列に変えたらどうなるだろうか。この時点で、f(a) の計算結果はもはやスカラーではない。結果はどうなるか。どう分析すればよいだろうか。

4. $f(x) = \sin(x)$ とする。f のグラフとその導関数 f' のグラフを描いよ。$f'(x) = \cos(x)$ であることは使わず、自動微分を用いて結果を得よ。

5. $f(x) = ((\log x^2) \cdot \sin x) + x^{-1}$ とする。x から $f(x)$ までの結果をたどる依存グラフを書いよ。

6. 連鎖律を用いて、前述の関数の導関数 $\frac{df}{dx}$ を計算し、先ほど構築した依存グラフ上の各項に対応付けよ。

7. グラフと中間導関数の結果が与えられたとき、勾配を計算する方法はいくつかある。x から f へ向かって一度計算し、もう一度 f から x に向かってたどって計算しよ。x から f への経路は一般に 順方向微分として知られ、f から x への経路は逆方向微分として知られている。

8. いつ順方向微分を使い、いつ逆方向微分を使うべきだろうか。ヒント: 必要な中間データ量、各ステップの並列化可能性、関係する行列やベクトルのサイズを考えよ。

5.6 確率と統計

いずれにせよ、機械学習は不確実性そのものである。教師あり学習では、既知の情報（特徴量）から未知の情報（ターゲット）を予測することを目指す。目的に応じて、最も可能性の高いターゲットの値を予測したり、ターゲットとの期待距離が最小になるような値を予測したりすることがある。また、単に値を予測するだけでなく、不確実性を定量化したい場合もある。たとえば、患者のデータから、その人が将来心臓発作を起こす可能性がどの程度あるかを知りたいといったケースである。教師なし学習では、しばしば不確実性が重要である。測定値の集合が異常かどうかを判断するには、関心のある母集団においてどのような値が観測される可能性が高いかを知ることが役立ちる。さらに、強化学習では、さまざまな環境で賢く行動するエージェントを開発したいと考える。そのためには、環境がどのように変化すると予想されるか、また利用可能な各行動に対してどのような報酬が得られると期待されるかを推論する必要がある。

確率は、不確実性の下での推論を扱う数学分野である。ある過程の確率モデルが与えられれば、さまざまな事象の起こりやすさについて推論できる。コイン投げのような繰り返し可能な事象の頻度を確率で表すことは、かなり異論の少ない考え方である。実際、頻度主義の研究者は、繰り返し可能な事象にのみ適用される確率の解釈を支持する。これに対して、ベイズ主義の研究者は、不確実性の下での推論を形式化するために、より広い意味で確率の言語を用いる。ベイズ確率には 2 つの特徴がある。(i) ダムが崩壊する確率はどれくらいか、といった繰り返し不可能な事象に対しても信念の度合いを割り当てること、(ii) 主観性である。ベイズ確率は、新しい証拠に照らして信念をどのように更新すべきかについて明確な規則を与えるが、異なる個人が異なる事前信念から出発することを許する。統計学は、データの収集と整理から始まり、そのデータを生成した背景にあるプロセスについて、どのような推論が可能かを逆向きに考えるための学問である。データセットを分析して、より広い母集団を特徴づけるかもしれないパターンを探すとき、私たちは統計的思考を用いている。確率と統計の研究に捧げられた講義、専攻、学位論文、職業、学科、企業、機関は数多くある。この節ではその表面をなぞるにすぎないが、モデル構築を始めるために必要な基礎を提供する。

```
%matplotlib inline
from d2l import torch as d2l
import random
import torch
from torch.distributions.multinomial import Multinomial
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx
```

(continues on next page)

(continued from previous page)

```
from mxnet.numpy.random import multinomial
import random
npx.set_np()
```

```
%matplotlib inline
from d2l import jax as d2l
import random
import jax
from jax import numpy as jnp
import numpy as np
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import random
import tensorflow as tf
from tensorflow_probability import distributions as tfd
```

5.6.1 簡単な例：コイン投げ

コインを投げると計画し、表（対して裏）がどれくらい起こりやすいかを定量化したいとする。コインが公平なら、両方の結果（表と裏）は同じ確率で起こりる。さらに、コインを n 回投げる予定なら、期待される表の割合は、期待される裏の割合とちょうど一致するはずである。これを直感的に見る一つの方法は対称性である。n_{h} 回の表と $n_{\mathrm{t}} = (n - n_{\mathrm{h}})$ 回の裏からなるあらゆる可能な結果に対して、n_{t} 回の表と n_{h} 回の裏からなる同じ確率の結果が存在する。これは、平均すると投げた回数の 1/2 が表、1/2 が裏になると期待できる場合にのみ可能である。もちろん、この実験を $n = 1000000$ 回の投げで何度も行っても、$n_{\mathrm{h}} = n_{\mathrm{t}}$ がぴったり成り立つ試行を一度も見ないかもしれない。

形式的には、量 1/2 は確率と呼ばれ、ここでは任意の 1 回の投げで表が出る確実性を表している。確率は、関心のある結果、すなわち事象に対して 0 から 1 の間の値を割り当てる。ここで関心のある事象は heads であり、対応する確率を $P(\text{heads})$ と表す。確率が 1 なら絶対確実を意味し（両面が表のイカサマコインを想像してほしい）、確率が 0 なら不可能を意味する（たとえば両面が裏なら）。頻度 n_{h}/n と n_{t}/n は確率ではなく、むしろ統計量である。確率は、データ生成プロセスの背後にある理論的な量である。ここでは、確率 0.5 はコインそのものが持つ物理的な性質と言える。これに対し統計量は、観測されたデータの関数として計算される経験的な量である。確率的・統計的な量への関心は切り離せないほど密接に結びついている。私たちはしばしば、データセットを与えると確率のようなモデルパラメータの推定値を生成する 推定量と呼ばれる特別な統計量を設計する。さらに、その推定量が一致性と呼ばれるよい性質を満たすなら、推定値は対応する確率に収束する。そして、そのように推定された確

率は、将来出会うかもしれない同じ母集団からのデータの統計的性質がどのようなものかを教えてくれる。

真の $P(\text{heads})$ を知らない実際のコインを見つけたとしよう。この量を統計的手法で調べるには、(i) いくつかのデータを収集し、(ii) 推定量を設計する必要がある。ここでのデータ取得は簡単である。コインを何度も投げて、すべての結果を記録すればよいのである。形式的には、ある基礎となるランダム過程から実現値を引き出すことをサンプリングと呼ぶ。ご想像のとおり、自然な推定量の一つは、観測された表の数を投げた総回数で割った比率である。

さて、そのコインが実際には公平、すなわち $P(\text{heads}) = 0.5$ だったとする。公平なコインの投げをシミュレートするには、任意の乱数生成器を呼び出せる。確率 0.5 の事象をサンプルする簡単な方法はいくつかある。たとえば Python の `random.random` は区間 $[0, 1]$ の数を返し、任意の部分区間 $[a, b] \subset [0, 1]$ に入る確率は $b - a$ に等しくなる。したがって、返された浮動小数点数が `0.5` より大きいかどうかを判定することで、`0` と `1` をそれぞれ確率 `0.5` で得られる。

```
num_tosses = 100
heads = sum([random.random() > 0.5 for _ in range(num_tosses)])
tails = num_tosses - heads
print("heads, tails: ", [heads, tails])
```

```
heads, tails:  [43, 57]
```

より一般には、可能な結果の数が有限な任意の変数（コイン投げやサイコロ投げのようなもの）からの複数回の抽出は、multinomial 関数を呼び出し、第 1 引数に抽出回数、第 2 引数に各可能な結果に対応する確率のリストを与えることでシミュレートできる。公平なコインを 10 回投げるには、確率ベクトル `[0.5, 0.5]` を割り当て、インデックス 0 を表、インデックス 1 を裏として解釈する。この関数は、可能な結果の数（ここでは 2）と同じ長さのベクトルを返し、最初の成分が表の出現回数、2 番目の成分が裏の出現回数を表す。

```
fair_probs = torch.tensor([0.5, 0.5])
Multinomial(100, fair_probs).sample()
```

```
tensor([58., 42.])
```

```
fair_probs = [0.5, 0.5]
multinomial(100, fair_probs)
```

```
[07:21:03] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
array([46, 54], dtype=int64)
```

```
fair_probs = [0.5, 0.5]
# jax.random does not have multinomial distribution implemented
np.random.multinomial(100, fair_probs)
```

```
array([54, 46])
```

```
fair_probs = tf.ones(2) / 2
tfd.Multinomial(100, fair_probs).sample()
```

```
WARNING:tensorflow:From /home/ci/.local/lib/python3.10/site-packages/
↪tensorflow_probability/python/internal/batched_rejection_sampler.py:102:␣
↪calling while_loop_v2 (from tensorflow.python.ops.control_flow_ops) with␣
↪back_prop=False is deprecated and will be removed in a future version.
Instructions for updating:
back_prop=False is deprecated. Consider using tf.stop_gradient instead.
Instead of:
results = tf.while_loop(c, b, vars, back_prop=False)
Use:
results = tf.nest.map_structure(tf.stop_gradient, tf.while_loop(c, b, vars))
```

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([49., 51.], dtype=float32)>
```

このサンプリング過程を実行するたびに、前回の結果とは異なるかもしれない新しい乱数値が得られる。投げた回数で割ると、データ中の各結果の頻度が得られる。これらの頻度は、それらが推定しようとしている確率と同様に、合計すると 1 になる。

```
Multinomial(100, fair_probs).sample() / 100
```

```
tensor([0.5200, 0.4800])
```

```
multinomial(100, fair_probs) / 100
```

```
array([0.53, 0.47])
```

```
np.random.multinomial(100, fair_probs) / 100
```

```
array([0.48, 0.52])
```

```
tfd.Multinomial(100, fair_probs).sample() / 100
```

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.51, 0.49],␣
↪dtype=float32)>
```

ここでは、シミュレートしたコインは公平であるにもかかわらず（私たち自身が確率 `[0.5, 0.5]` を設定した）、表と裏の回数は一致しないかもしれない。それは、比較的少数のサンプルしか取っていないからである。もしシミュレーションを自分で実装しておらず、結果だけを見ていたとしたら、コインが少し不公平なのか、それとも 1/2 からのずれが単にサンプルサイズの小ささによるものなのか、どうやって判断すればよいだろうか？10,000 回投げをシミュレートするとどうなるか見てみよう。

```
counts = Multinomial(10000, fair_probs).sample()
counts / 10000
```

```
tensor([0.4953, 0.5047])
```

```
counts = multinomial(10000, fair_probs).astype(np.float32)
counts / 10000
```

```
array([0.4952, 0.5048])
```

```
counts = np.random.multinomial(10000, fair_probs).astype(np.float32)
counts / 10000
```

```
array([0.4904, 0.5096], dtype=float32)
```

```
counts = tfd.Multinomial(10000, fair_probs).sample()
counts / 10000
```

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.5038, 0.4962],␣
↪dtype=float32)>
```

一般に、繰り返し事象（コイン投げのようなもの）の平均については、繰り返し回数が増えるにつれて、推定値は真の基礎確率に収束することが保証される。この現象の数学的定式化は大数の法則と呼ばれ、中心極限定理は、多くの状況でサンプルサイズ n が増えるにつれてこれらの誤差が $(1/\sqrt{n})$ の速度で減少すべきことを示す。投げる回数を 1 回から 10,000 回まで増やしたときに推定値がどのように変化するかを調べて、もう少し直感を得よう。

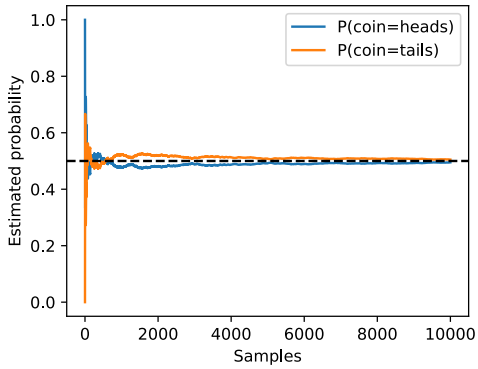
```
counts = Multinomial(1, fair_probs).sample((10000,))
cum_counts = counts.cumsum(dim=0)
```

(continues on next page)

(continued from previous page)

```
estimates = cum_counts / cum_counts.sum(dim=1, keepdims=True)
estimates = estimates.numpy()

d2l.set_figsize((4.5, 3.5))
d2l.plt.plot(estimates[:, 0], label=("P(coin=heads)"))
d2l.plt.plot(estimates[:, 1], label=("P(coin=tails)"))
d2l.plt.axhline(y=0.5, color='black', linestyle='dashed')
d2l.plt.gca().set_xlabel('Samples')
d2l.plt.gca().set_ylabel('Estimated probability')
d2l.plt.legend();
```




```
counts = multinomial(1, fair_probs, size=10000)
cum_counts = counts.astype(np.float32).cumsum(axis=0)
estimates = cum_counts / cum_counts.sum(axis=1, keepdims=True)
```

```
counts = np.random.multinomial(1, fair_probs, size=10000).astype(np.float32)
cum_counts = counts.cumsum(axis=0)
estimates = cum_counts / cum_counts.sum(axis=1, keepdims=True)
```

```
counts = tfd.Multinomial(1, fair_probs).sample(10000)
cum_counts = tf.cumsum(counts, axis=0)
estimates = cum_counts / tf.reduce_sum(cum_counts, axis=1, keepdims=True)
estimates = estimates.numpy()
```

各実線はコインの 2 つの値の一方に対応し、各実験グループの後にその値が出る確率の推定値を示す。黒の破線は真の基礎確率を表す。実験を重ねてより多くのデータを得るにつれて、曲線は真の確率へ収束していきる。統計学者を悩ませる、より高度な問いの形がすでに見え始めているかもしれない。この収束はどれくらい速く起こるのか？もし同じ工場で製造されたコインをすでに多数試していたら、その情報をどのように取り込めるだろうか？

5.6.2 より形式的な扱い

ここまででかなり進んだ。確率モデルを立て、合成データを生成し、統計的推定量を実行し、収束を経験的に評価し、誤差指標（ずれの確認）を報告した。しかし、さらに先へ進むには、より厳密である必要がある。

ランダム性を扱うとき、可能な結果の集合を $\mathcal{S}$ と表し、これを標本空間または結果空間と呼ぶ。ここで各要素は異なる可能な結果である。1 回のコイン投げの場合、$\mathcal{S} = \{\text{heads}, \text{tails}\}$ である。1 回のサイコロ投げでは、$\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ である。2 枚のコインを投げると、可能な結果は $\{(\text{heads}, \text{heads}), (\text{heads}, \text{tails}), (\text{tails}, \text{heads}), (\text{tails}, \text{tails})\}$ である。事象は標本空間の部分集合である。たとえば、「1 枚目のコイン投げが表になる」という事象は集合 $\{(\text{heads}, \text{heads}), (\text{heads}, \text{tails})\}$ に対応する。ランダム実験の結果 z が $z \in \mathcal{A}$ を満たすとき、事象 $\mathcal{A}$ が起こったと言える。1 回のサイコロ投げについて、「5 が出る」（$\mathcal{A} = \{5\}$）と「奇数が出る」（$\mathcal{B} = \{1, 3, 5\}$）という事象を定義できる。この場合、サイコロが 5 なら、$\mathcal{A}$ と $\mathcal{B}$ の両方が起こったと言える。一方、$z = 3$ なら、$\mathcal{A}$ は起こらず、$\mathcal{B}$ は起こりる。

確率関数は事象を実数値へ写像する $P : \mathcal{A} \subseteq \mathcal{S} \to [0, 1]$。与えられた標本空間 $\mathcal{S}$ における事象 $\mathcal{A}$ の確率を $P(\mathcal{A})$ と表し、次の性質を持つ。

- 任意の事象 $\mathcal{A}$ の確率は非負の実数である。すなわち $P(\mathcal{A}) \geq 0$;
- 標本空間全体の確率は 1 である。すなわち $P(\mathcal{S}) = 1$;
- 互いに排反な可算個の事象列 $\mathcal{A}_1, \mathcal{A}_2, \ldots$（すなわち、すべての $i \neq j$ について $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$）に対して、それらのどれかが起こる確率は各確率の和に等しい。すなわち $P(\bigcup_{i=1}^{\infty} \mathcal{A}_i) = \sum_{i=1}^{\infty} P(\mathcal{A}_i)$。

確率論のこれらの公理は、Kolmogorov (1933) によって提案され、多くの重要な帰結を素早く導くことができる。たとえば、任意の事象 $\mathcal{A}$ またはその補集合 $\mathcal{A}'$ が起こる確率は 1 であることがすぐに従います（$\mathcal{A} \cup \mathcal{A}' = \mathcal{S}$ だからです）。また、$P(\emptyset) = 0$ も証明できる。なぜなら $1 = P(\mathcal{S} \cup \mathcal{S}') = P(\mathcal{S} \cup \emptyset) = P(\mathcal{S}) + P(\emptyset) = 1 + P(\emptyset)$ だからである。したがって、任意の事象 $\mathcal{A}$ かつその補集合 $\mathcal{A}'$ が同時に起こる確率は $P(\mathcal{A} \cap \mathcal{A}') = 0$ である。非形式的に言えば、これは不可能な事象の起こる確率はゼロであることを意味する。

5.6.3 確率変数

サイコロの目が奇数になることや、1 回目のコイン投げが表になることのような事象について話すとき、私たちは確率変数の概念を用いていた。形式的には、確率変数は基礎となる標本空間から（場合によっては多数の）値の集合への写像である。確率変数と標本空間はどちらも結果の集合なので、何が違うのか疑問に思うかもしれない。重要なのは、確率変数は生の標本空間よりもはるかに粗くできるということである。たとえば、基礎となる標本空間が 0 と 1 の線分上の点のように無限であっても、「0.5 より大きい」といった二値の確率変数を定義できる。さらに、複数の確率変数が同じ基礎標本空間を共有することもある。たとえば、「自宅の警報装置が鳴るかどうか」と「家に泥棒が入ったかどうか」は、どちらも同じ基礎標本空間を共有する二値の確率変数である。したがって、ある確率変数が取る値を知ると、別の確率変数が取りうる値について

何かを知ることができる。警報が鳴ったと分かれば、家に泥棒が入った可能性が高いと疑うかもしれない。

確率変数が取る各値は、基礎となる標本空間の部分集合に対応する。したがって、確率変数 X が値 v を取る事象、すなわち $X = v$ は事象であり、$P(X = v)$ はその確率を表す。この記法はときに煩雑になるため、文脈が明らかな場合には記法を流用できる。たとえば、$P(X)$ を広く X の分布、すなわち X が任意の値を取る確率を与える関数を指すのに使うことがある。また、$P(X,Y) = P(X)P(Y)$ のような式を書くこともある。これは、確率変数 X と Y が取りうるすべての値に対して成り立つ文を簡潔に表すためのもので、すなわちすべての i, j について $P(X = i \text{ and } Y = j) = P(X = i)P(Y = j)$ が成り立つという意味である。さらに、確率変数が文脈から明らかなときには $P(v)$ と書いて記法を流用することもある。確率論における事象は標本空間の結果の集合なので、確率変数が取る値の範囲を指定できる。たとえば、$P(1 \leq X \leq 3)$ は事象 $\{1 \leq X \leq 3\}$ の確率を表す。

離散確率変数（コイン投げやサイコロ投げのようなもの）と、連続確率変数（母集団から無作為に抽出した人の体重や身長のようなもの）の間には、微妙な違いがあることに注意しよ。この場合、私たちは誰かの正確な身長をそれほど気にすることはほとんどない。さらに、十分に精密に測定すれば、地球上でまったく同じ身長の人は一人もいないことが分かるだろう。実際、十分に細かく測れば、起床時と就寝時で同じ身長になることもない。誰かの身長が 1.801392782910287192 メートルである正確な確率を問うことには、ほとんど意味がない。むしろ通常は、誰かの身長がたとえば 1.79 メートルから 1.81 メートルの間に入るかどうかを知ることの方が重要である。このような場合には、確率密度を扱いる。ちょうど 1.80 メートルである確率はないが、密度はゼロではない。区間に割り当てられる確率を求めるには、その区間上で密度を積分しなければならない。

5.6.4 複数の確率変数

前節を通り抜けるだけでも、複数の確率変数の相互作用を含む記述を避けられなかったことに気づいたかもしれない（$P(X,Y) = P(X)P(Y)$ を思い出してほしい）。機械学習の大部分は、このような関係を扱っている。ここでの標本空間は関心のある母集団、たとえば企業と取引する顧客、インターネット上の写真、あるいは生物学者に知られているタンパク質などである。各確率変数は、異なる属性の（未知の）値を表す。母集団から個体をサンプルするたびに、各確率変数の実現値を観測する。確率変数が取る値は、重なりうる、部分的に重なりうる、あるいは完全に互いに素な標本空間の部分集合に対応するため、ある確率変数の値を知ると、別の確率変数が取りうる値についての信念を更新することになる。患者が病院に来て、呼吸困難があり、嗅覚を失っていることが観測されたなら、呼吸困難もなく嗅覚も正常な場合よりも、COVID-19 にかかっている可能性が高いと考える。

複数の確率変数を扱うとき、変数が同時に取りうる値のあらゆる組合せに対応する事象を構成できる。これらの各組合せ（たとえば $A = a$ かつ $B = b$）に確率を割り当てる関数は 同時確率関数と呼ばれ、対応する標本空間の部分集合の共

通部分に割り当てられた確率をそのまま返す。確率変数 A と B がそれぞれ値 a と b を取る事象に割り当てられる 同時確率は $P(A = a, B = b)$ と表され、ここでカンマは「かつ」を意味する。任意の値 a と b に対して、次が成り立つことに注意しよ。

$$P(A = a, B = b) \leq P(A = a) \text{ and } P(A = a, B = b) \leq P(B = b), \tag{5.6.1}$$

なぜなら、$A = a$ と $B = b$ が起こるには、$A = a$ が起こり、かつ $B = b$ も起こらなければならないからである。興味深いことに、同時確率は、これらの確率変数について確率論的な意味で知りうるすべてを教えてくれ、個々の分布 $P(A)$ と $P(B)$ を復元することを含む多くの有用な量を導くために使える。$P(A = a)$ を復元するには、確率変数 B が取りうるすべての値 v について $P(A = a, B = v)$ を足し合わせればよいのである。$P(A = a) = \sum_v P(A = a, B = v)$。

比 $\frac{P(A=a,B=b)}{P(A=a)} \leq 1$ は非常に重要であることが分かりる。これは条件付き確率と呼ばれ、“|”記号で表される。

$$P(B = b \mid A = a) = P(A = a, B = b)/P(A = a). \tag{5.6.2}$$

これは、$A = a$ が起こったという事実を条件にしたときの、事象 $B = b$ に対応する新しい確率を教えてくれる。この条件付き確率は、標本空間のうち $A = a$ に対応する部分だけに注意を向け、その後、すべての確率の和が 1 になるように再正規化したものと考えられる。条件付き確率は実際には通常の確率にすぎず、したがって、すべての項を同じ事象で条件づけて同じ標本空間に注意を限定する限り、すべての公理に従いる。たとえば、互いに素な事象 $\mathcal{B}$ と $\mathcal{B}'$ に対して、$P(\mathcal{B} \cup \mathcal{B}' \mid A = a) = P(\mathcal{B} \mid A = a) + P(\mathcal{B}' \mid A = a)$ が成り立ちる。

条件付き確率の定義を用いると、有名な結果であるベイズの定理を導ける。定義により、$P(A, B) = P(B \mid A)P(A)$ かつ $P(A, B) = P(A \mid B)P(B)$ である。両式を組み合わせると $P(B \mid A)P(A) = P(A \mid B)P(B)$ となり、したがって

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}. \tag{5.6.3}$$

この単純な式は、条件づけの順序を逆転できるため、深い意味を持つ。$P(B \mid A)$、$P(A)$、$P(B)$ をどのように推定するかが分かれば、$P(A \mid B)$ を推定できる。一方を直接推定する方が他方より簡単なことがよくあり、そのときベイズの定理が助けになる。たとえば、ある病気に対する症状の有病率と、それぞれ病気および症状全体の有病率が分かっていれば、症状に基づいてその人が病気である可能性を決定できる。場合によっては、症状の有病率のような $P(B)$ に直接アクセスできないこともある。この場合、ベイズの定理の簡略版が役立ちる。

$$P(A \mid B) \propto P(B \mid A)P(A). \tag{5.6.4}$$

$P(A \mid B)$ は 1 に正規化されなければならない、すなわち $\sum_a P(A = a \mid B) = 1$ であることが分かっているので、次を計算できる。

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{\sum_a P(B \mid A = a)P(A = a)}. \tag{5.6.5}$$

ベイズ統計では、観測者は利用可能な仮説の妥当性についてのある（主観的な）事前信念を 事前分布 $P(H)$ に符号化して持ち、仮説の各クラスに対して収集された証拠の各値を観測する確率を表す 尤度関数 $P(E \mid H)$ を持つと考える。ベイズの定理は、利用可能な証拠 E に照らして初期の事前 $P(H)$ をどのように更新し、事後信念 $P(H \mid E) = \frac{P(E|H)P(H)}{P(E)}$ を得るかを教えてくれるものとして解釈される。非形式的には、「事後は事前に尤度を掛けて証拠で割ったもの」と表現できる。ここで証拠 $P(E)$ はすべての仮説で同じなので、仮説全体で正規化するだけで済みる。

$\sum_a P(A = a \mid B) = 1$ は、確率変数を周辺化することも可能にする。つまり、$P(A, B)$ のような同時分布から変数を取り除けるのである。実際、

$$\sum_a P(B \mid A = a)P(A = a) = \sum_a P(B, A = a) = P(B). \tag{5.6.6}$$

独立性もまた、統計学の多くの重要な考え方の基盤をなす根本的に重要な概念である。要するに、2 つの変数は、A の値で条件づけても B に対応する確率分布に変化が生じず、その逆も同様であれば独立である。より形式的には、$A \perp B$ と表される独立性は、$P(A \mid B) = P(A)$、したがって $P(A, B) = P(A \mid B)P(B) = P(A)P(B)$ を要求する。独立性はしばしば適切な仮定である。たとえば、確率変数 A が 1 枚目の公平なコインを投げた結果を表し、確率変数 B が別のコインを投げた結果を表すなら、A が表だったかどうかは B が表になる確率に影響すべきではない。

独立性は、基礎となる分布からのデータの連続した抽出の間で成り立つとき（強い統計的結論を導けるため）、あるいはデータ中のさまざまな変数の間で成り立つとき（その独立構造を符号化したより単純なモデルを扱えるため）、特に有用である。一方で、確率変数間の依存関係を推定することこそが学習の目的そのものであることも少なくない。症状から病気の確率を推定したいのは、病気と症状が独立ではないと考えているからである。

条件付き確率も正しい確率であるため、独立と従属の概念はそれにも適用されることに注意しよ。3 つ目の変数 C が与えられたとき、2 つの確率変数 A と B が条件付き独立であるとは、$P(A, B \mid C) = P(A \mid C)P(B \mid C)$ が成り立つことと同値である。興味深いことに、2 つの変数は一般には独立でも、3 つ目の変数で条件づけると従属になることがある。これはしばしば、2 つの確率変数 A と B が 3 つ目の変数 C の原因に対応するときに起こりる。たとえば、骨折と肺がんは一般集団では独立かもしれないが、病院にいることを条件にすると、骨折は肺がんと負の相関を持つかもしれない。それは、骨折がその人が病院にいる理由を説明し尽くすため、肺がんのために入院している可能性を下げるからである。

逆に、2 つの従属な確率変数が 3 つ目の変数で条件づけると独立になることもある。これは、もともと無関係な 2 つの事象に共通の原因があるときによく起こりる。小学生の間では、靴のサイズと読解力は強く相関しているが、年齢で条件づけるとこの相関は消える。

5.6.5 例

技能を試してみよう。医師が患者に HIV 検査を行うとする。この検査はかなり正確で、患者が健康なのに病気と報告される場合、すなわち健康な患者が陽性と判定される場合にのみ、1% の確率で失敗する。さらに、患者が実際に HIV に感染している場合には、見逃すことはない。診断を $D_1 \in \{0,1\}$ で表し（0 は陰性、1 は陽性）、HIV の状態を $H \in \{0,1\}$ で表す。

条件付き確率	$H=1$	$H=0$
$P(D_1 = 1 \mid H)$	1	0.01
$P(D_1 = 0 \mid H)$	0	0.99

列の和はすべて 1 であるが、行の和はそうではないことに注意しよ。これは条件付き確率だからである。検査結果が陽性だったときに患者が HIV に感染している確率、すなわち $P(H = 1 \mid D_1 = 1)$ を計算しよう。直感的には、これは病気の一般的な有病率に依存する。なぜなら、それが偽陽性の数に影響するからである。母集団では病気がかなり少ないとし、たとえば $P(H = 1) = 0.0015$ とする。ベイズの定理を適用するには、周辺化を用いて次を求める必要がある。

$$\begin{aligned} P(D_1 = 1) =& P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1) \\ =& P(D_1 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1 \mid H = 1)P(H = 1) \\ =& 0.011485. \end{aligned} \tag{5.6.7}$$

これにより、

$$P(H = 1 \mid D_1 = 1) = \frac{P(D_1 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1)} = 0.1306. \tag{5.6.8}$$

言い換えると、検査はかなり正確であるにもかかわらず、患者が実際に HIV に感染している確率はわずか 13.06% しかない。ご覧のとおり、確率は直感に反することがある。このような恐ろしい知らせを受けた患者はどうすべきだろうか？おそらく、患者は医師にもう一度検査をしてもらい、はっきりさせたいと頼むだろう。2 回目の検査は性質が異なり、1 回目ほど良くはない。

条件付き確率	$H=1$	$H=0$
$P(D_2 = 1 \mid H)$	0.98	0.03
$P(D_2 = 0 \mid H)$	0.02	0.97

残念ながら、2 回目の検査も陽性でした。条件付き独立を仮定して、ベイズの定理を適用するために必要な確率を計算しよう。

$$\begin{aligned} P(D_1 = 1, D_2 = 1 \mid H = 0) &= P(D_1 = 1 \mid H = 0)P(D_2 = 1 \mid H = 0) = 0.0003, \\ P(D_1 = 1, D_2 = 1 \mid H = 1) &= P(D_1 = 1 \mid H = 1)P(D_2 = 1 \mid H = 1) = 0.98. \end{aligned} \tag{5.6.9}$$

これで周辺化を適用し、両方の検査が陽性となる確率を得られる。

$$\begin{aligned}
&P(D_1 = 1, D_2 = 1) \\
&= P(D_1 = 1, D_2 = 1, H = 0) + P(D_1 = 1, D_2 = 1, H = 1) \\
&= P(D_1 = 1, D_2 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1) \\
&= 0.00176955.
\end{aligned} \tag{5.6.10}$$

最後に、両方の検査が陽性であるときに患者が HIV に感染している確率は

$$P(H = 1 \mid D_1 = 1, D_2 = 1) = \frac{P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1, D_2 = 1)} = 0.8307. \tag{5.6.11}$$

つまり、2 回目の検査によって、すべてが順調ではないという確信をかなり高めることができた。2 回目の検査は 1 回目よりかなり精度が低いにもかかわらず、それでも推定を大きく改善した。2 つの検査が互いに条件付き独立であるという仮定は、より正確な推定を得るうえで決定的でした。極端な例として、同じ検査を 2 回行う場合を考えてみよう。この状況では、2 回とも同じ結果になると期待されるため、同じ検査をもう一度行っても追加の洞察は得られない。鋭い読者は、診断が見え透いた分類器のように振る舞っており、患者が健康かどうかを判断する能力はより多くの特徴量（検査結果）を得るにつれて高まることに気づいたかもしれない。

5.6.6 期待値

しばしば、意思決定には個々の事象に割り当てられた確率を見るだけでなく、それらを組み合わせて、指針を与えてくれる有用な集約量を作ることが必要である。たとえば、確率変数が連続的なスカラー値を取るとき、私たちは平均的にどの値が期待されるかを知りたいことがよくある。この量は形式的には期待値と呼ばれる。投資を行う場合、最初に関心を持つ量は、起こりうるすべての結果にわたって平均した期待収益かもしれない（適切な確率で重みづけする）。たとえば、50% の確率で投資が完全に失敗し、40% の確率で 2× のリターンを生み、10% の確率で 10× のリターンを生むとする。期待収益を計算するには、すべてのリターンについて、それぞれに起こる確率を掛けて足し合わせる。すると期待値は $0.5 \cdot 0 + 0.4 \cdot 2 + 0.1 \cdot 10 = 1.8$ となる。したがって、期待収益は 1.8× である。

一般に、確率変数 X の期待値（または平均）は次のように定義される。

$$E[X] = E_{x \sim P}[x] = \sum_x xP(X = x). \tag{5.6.12}$$

同様に、密度に対しては $E[X] = \int x \, dp(x)$ を得る。ときには、x のある関数の期待値に関心がある。これらの期待値は次のように計算できる。

$$E_{x \sim P}[f(x)] = \sum_x f(x)P(x) \text{ and } E_{x \sim P}[f(x)] = \int f(x)p(x) \, dx \tag{5.6.13}$$

それぞれ離散確率と密度に対してである。上の投資の例に戻ると、f はリター

ンに対応する効用（幸福度）かもしれない。行動経済学では、人々は 1 ドルを稼ぐことで得られる効用よりも、お金を失うことに対してより大きな不効用を感じることが長く指摘されてきた。さらに、お金の価値はしばしば準線形である。10 万ドルを持つこととゼロドルであることの違いは、家賃を払えるか、十分に食べられるか、質の高い医療を受けられるか、それともホームレスとして苦しむかの違いになりえる。一方で、20 万ドルと 10 万ドルの差による利益はそれほど劇的ではない。このような考え方は、「お金の効用は対数的である」という決まり文句を動機づける。

総損失に対応する効用が –1 で、リターン 1、2、10 に対応する効用がそれぞれ 1、2、4 だとすると、投資の期待幸福度は $0.5 \cdot (-1) + 0.4 \cdot 2 + 0.1 \cdot 4 = 0.7$（期待効用損失 30%）になる。もし本当にこれがあなたの効用関数なら、お金を銀行に預けておくのが最善かもしれない。

金融上の意思決定では、投資がどれだけリスクが高いかも測りたいかもしれない。ここでは期待値だけでなく、実際の値がこの値に対してどれだけ変動しやすいかに関心がある。実際の値と期待値の差の期待値をそのまま取ることはできない点に注意しよ。差の期待値は期待値の差だからである。すなわち、$E[X - E[X]] = E[X] - E[E[X]] = 0$ である。しかし、この差の任意の非負関数の期待値を見ることはできる。確率変数の分散は、二乗差の期待値を見ることで計算される。

$$\mathrm{Var}[X] = E\left[(X - E[X])^2\right] = E[X^2] - E[X]^2. \tag{5.6.14}$$

ここで等式は、$(X - E[X])^2 = X^2 - 2XE[X] + E[X]^2$ を展開し、各項の期待値を取ることで得られる。分散の平方根は、標準偏差と呼ばれる別の有用な量である。分散と標準偏差は同じ情報を伝えますが（どちらか一方から他方を計算できる）、標準偏差には、元の確率変数が表す量と同じ単位で表されるという利点がある。

最後に、確率変数の関数の分散は同様に次のように定義される。

$$\mathrm{Var}_{x\sim P}[f(x)] = E_{x\sim P}[f^2(x)] - E_{x\sim P}[f(x)]^2. \tag{5.6.15}$$

投資の例に戻ると、投資の分散を計算できる。それは $0.5 \cdot 0 + 0.4 \cdot 2^2 + 0.1 \cdot 10^2 - 1.8^2 = 8.36$ である。あらゆる意味で、これはリスクの高い投資である。数学的慣習として、平均と分散はしばしば μ と σ^2 で表されることに注意しよ。これは特に、ガウス分布をパラメータ化するときによく使われる。

スカラー確率変数に対して期待値と分散を導入したのと同様に、ベクトル値の確率変数にも同じことができる。期待値は要素ごとに適用できるので簡単である。たとえば、$\boldsymbol{\mu} \stackrel{\mathrm{def}}{=} E_{\mathbf{x}\sim P}[\mathbf{x}]$ の各成分は $\mu_i = E_{\mathbf{x}\sim P}[x_i]$ である。共分散はより複雑である。これは、確率変数とその平均との差の外積の期待値として定義する。

$$\boldsymbol{\Sigma} \stackrel{\mathrm{def}}{=} \mathrm{Cov}_{\mathbf{x}\sim P}[\mathbf{x}] = E_{\mathbf{x}\sim P}\left[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top\right]. \tag{5.6.16}$$

この行列 $\boldsymbol{\Sigma}$ は共分散行列と呼ばれる。その効果を理解する簡単な方法は、$\mathbf{x}$ と

同じ大きさのベクトル $\mathbf{v}$ を考えることである。すると次が成り立ちる。

$$\mathbf{v}^\top \boldsymbol{\Sigma} \mathbf{v} = E_{\mathbf{x}\sim P}\left[\mathbf{v}^\top(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^\top \mathbf{v}\right] = \mathrm{Var}_{x\sim P}[\mathbf{v}^\top \mathbf{x}]. \tag{5.6.17}$$

したがって、$\boldsymbol{\Sigma}$ は、$\mathbf{x}$ の任意の線形関数の分散を単純な行列積で計算できるようにする。非対角成分は各成分の相関の強さを示す。0 なら相関なし、より大きな正の値ならより強く相関していることを意味する。

5.6.7 議論

機械学習では、不確実なことがたくさんある。入力が与えられたときのラベルの値について不確実であることもあれば、推定されたパラメータの値について不確実であることもある。さらには、運用時に到着するデータが学習データと同じ分布から来ているかどうかさえ不確実であることがある。

偶然的不確実性とは、問題に内在する不確実性であり、観測変数では説明できない真のランダム性によるものを指す。認識的不確実性とは、モデルのパラメータに関する不確実性であり、より多くのデータを集めることで減らせると期待できる種類の不確実性である。コインが表になる確率については認識的不確実性を持つかもしれないが、その確率が分かった後でも、将来の任意の投げの結果については偶然的不確実性が残る。誰かが公平なコインを投げるのをどれだけ長く見ていても、次の投げが表になると 50% より高くも低くも確信することはできない。これらの用語は機械的モデリングに由来する（この側面の不確実性定量化[45] については、たとえば Der Kiureghian and Ditlevsen (2009) のレビューを参照されたい）。ただし、これらの用語はやや言葉の流用であることに注意する価値がある。*epistemic* という語は知識に関するあらゆるものを指すため、哲学的な意味では、あらゆる不確実性は認識的不確実性である。

[45] 

未知の確率分布からデータをサンプリングすることで、データ生成分布のパラメータを推定するために使える情報が得られることを見た。とはいえ、これが可能になる速度はかなり遅いことがある。コイン投げの例（および多くの他の例）では、サンプルサイズ n に対して $1/\sqrt{n}$ の速度で収束する推定量を設計する以上のことはできない。これは、10 個の観測から 1000 個の観測へ増やすと（通常は十分達成可能な課題です）不確実性が 10 分の 1 に減る一方で、次の 1000 個の観測は比較的あまり役に立たず、1.41 倍の削減しかもたらさないことを意味する。これは機械学習における持続的な特徴である。しばしば容易な改善はあるが、さらに改善するには非常に大量のデータ、そしてしばしば膨大な計算量が必要になる。大規模言語モデルに関するこの事実の実証的レビューについては Revels *et al.* (2016) を参照しよ。

また、統計モデリングのための言語と道具も洗練された。その過程で、条件付き確率と、統計学で最も重要な方程式の一つであるベイズの定理について学んだ。これは、データが伝える情報を、観測 B がパラメータの選択 A にどれだけ適合するかを表す尤度項 $P(B \mid A)$ と、そもそも特定の A の選択がどれほど妥当だったかを支配する事前確率 $P(A)$ によって切り分けるための有効な道具である。特に、この規則が、検査の有効性と病気そのものの有病率（すなわち事

前）に基づいて、診断に確率を割り当てるためにどのように適用できるかを見た。

最後に、特定の確率分布の効果についての最初の非自明な問い、すなわち期待値と分散を導入した。確率分布には線形期待値や二次期待値以外にも多くのものがあるが、この 2 つだけでも分布の起こりうる振る舞いについてかなりの知識を与えてくれる。たとえば、チェビシェフの不等式[46] は $P(|X-\mu| \geq k\sigma) \leq 1/k^2$ を述べる。ここで μ は期待値、σ^2 は分散、$k > 1$ は任意に選べる信頼パラメータである。これは、分布からのサンプルが少なくとも 50% の確率で期待値を中心とする $[-\sqrt{2}\sigma, \sqrt{2}\sigma]$ 区間内に入ることを教えてくれる。

[46]

5.6.8 演習

1. より多くのデータを観測することで、結果に関する不確実性を任意に低いレベルまで減らせる例を挙げよ。

2. より多くのデータを観測しても、不確実性がある点までしか減らず、その後はそれ以上減らない例を挙げよ。なぜそうなるのか、またその点がどこにあると予想されるかを説明せよ。

3. コイン投げにおける平均への収束を経験的に示した。n 個のサンプルを得た後に表が出る確率の推定値の分散を計算せよ。

   1. 分散は観測数に対してどのようにスケールするか。

   2. チェビシェフの不等式を用いて期待値からのずれを上界づけよ。

   3. それは中心極限定理とどう関係するか。

4. 平均 0、分散 1 の確率分布から m 個のサンプル x_i を得ると仮定する。平均 $z_m \stackrel{\text{def}}{=} m^{-1} \sum_{i=1}^{m} x_i$ を計算せよ。各 z_m に対して独立にチェビシェフの不等式を適用できるか。なぜか。

5. 確率 $P(\mathcal{A})$ と $P(\mathcal{B})$ を持つ 2 つの事象が与えられたとき、$P(\mathcal{A} \cup \mathcal{B})$ と $P(\mathcal{A} \cap \mathcal{B})$ の上界と下界を求めよ。ヒント：ベン図[47] を使って状況を図示せよ。 [47]

6. A、B、C という確率変数列があり、B は A のみに依存し、C は B のみに依存すると仮定する。このとき同時確率 $P(A, B, C)$ を簡単化できるか。ヒント：これはマルコフ連鎖[48] である。 [48]

7. Section 5.6.5 では、2 つの検査結果が独立ではないと仮定せよ。特に、各検査単独の偽陽性率が 10%、偽陰性率が 1% であるとする。すなわち、$P(D = 1 \mid H = 0) = 0.1$ かつ $P(D = 0 \mid H = 1) = 0.01$ とする。さらに、$H = 1$（感染）では検査結果は条件付き独立、すなわち $P(D_1, D_2 \mid H = 1) = P(D_1 \mid H = 1)P(D_2 \mid H = 1)$ だが、健康な患者では結果が $P(D_1 = D_2 = 1 \mid H = 0) = 0.02$ を通じて結びついていると仮定する。

   1. ここまでの情報に基づいて、$H = 0$ のときの D_1 と D_2 の同時確率表を作成せよ。

2. 1 回目の検査が陽性だった後に患者が病気（$H = 1$）である確率を導出せよ。以前と同じ基準確率 $P(H = 1) = 0.0015$ を仮定してよい。
3. 両方の検査が陽性だった後に患者が病気（$H = 1$）である確率を導出せよ。

8. あなたが投資銀行の資産運用担当者で、投資先として株式 s_i を選べるとする。ポートフォリオは各株式に対して重み α_i を用いて合計が 1 になる必要がある。株式の平均収益率は $\boldsymbol{\mu} = E_{\mathbf{s}\sim P}[\mathbf{s}]$、共分散は $\boldsymbol{\Sigma} = \mathrm{Cov}_{\mathbf{s}\sim P}[\mathbf{s}]$ である。
   1. 与えられたポートフォリオ $\boldsymbol{\alpha}$ の期待収益を計算せよ。
   2. ポートフォリオの収益を最大化したいなら、投資をどのように選ぶべきか。
   3. ポートフォリオの分散を計算せよ。
   4. 分散に上限制約を課しつつ収益を最大化する最適化問題を定式化せよ。これはノーベル賞を受賞したマルコヴィッツのポートフォリオ[49] です (Mangram, 2013)。これを解くには二次計画法ソルバが必要であり、本書の範囲をはるかに超えている。

[49]

5.7 ドキュメント

すべての PyTorch の関数やクラスをここで紹介することは到底できないし、その情報はすぐに古くなってしまうかもしれないが、API ドキュメント[50] や追加のチュートリアル[51] と例がそのような文書を提供している。この節では、PyTorch の API を調べるための指針を示す。

[50]

[51]

```
import torch
```

```
from mxnet import np
```

```
import jax
```

```
import tensorflow as tf
```

5.7.1 モジュール内の関数とクラス

モジュール内でどの関数やクラスを呼び出せるかを知るには、`dir` 関数を使う。たとえば、乱数生成のためのモジュール内のすべての属性を問い合わせることができる。

```
print(dir(torch.distributions))
```

```
['AbsTransform', 'AffineTransform', 'Bernoulli', 'Beta', 'Binomial',
↪'CatTransform', 'Categorical', 'Cauchy', 'Chi2', 'ComposeTransform',
↪'ContinuousBernoulli', 'CorrCholeskyTransform',
↪'CumulativeDistributionTransform', 'Dirichlet', 'Distribution', 'ExpTransform
↪', 'Exponential', 'ExponentialFamily', 'FisherSnedecor', 'Gamma', 'Geometric
↪', 'Gumbel', 'HalfCauchy', 'HalfNormal', 'Independent', 'IndependentTransform
↪', 'Kumaraswamy', 'LKJCholesky', 'Laplace', 'LogNormal', 'LogisticNormal',
↪'LowRankMultivariateNormal', 'LowerCholeskyTransform', 'MixtureSameFamily',
↪'Multinomial', 'MultivariateNormal', 'NegativeBinomial', 'Normal',
↪'OneHotCategorical', 'OneHotCategoricalStraightThrough', 'Pareto', 'Poisson',
↪ 'PositiveDefiniteTransform', 'PowerTransform', 'RelaxedBernoulli',
↪'RelaxedOneHotCategorical', 'ReshapeTransform', 'SigmoidTransform',
↪'SoftmaxTransform', 'SoftplusTransform', 'StackTransform',
↪'StickBreakingTransform', 'StudentT', 'TanhTransform', 'Transform',
↪'TransformedDistribution', 'Uniform', 'VonMises', 'Weibull', 'Wishart', '__
↪all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '_
↪_name__', '__package__', '__path__', '__spec__', 'bernoulli', 'beta',
↪'biject_to', 'binomial', 'categorical', 'cauchy', 'chi2', 'constraint_
↪registry', 'constraints', 'continuous_bernoulli', 'dirichlet', 'distribution
↪', 'exp_family', 'exponential', 'fishersnedecor', 'gamma', 'geometric',
↪'gumbel', 'half_cauchy', 'half_normal', 'identity_transform', 'independent',
↪'kl', 'kl_divergence', 'kumaraswamy', 'laplace', 'lkj_cholesky', 'log_normal
↪', 'logistic_normal', 'lowrank_multivariate_normal', 'mixture_same_family',
↪'multinomial', 'multivariate_normal', 'negative_binomial', 'normal', 'one_
↪hot_categorical', 'pareto', 'poisson', 'register_kl', 'relaxed_bernoulli',
↪'relaxed_categorical', 'studentT', 'transform_to', 'transformed_distribution
↪', 'transforms', 'uniform', 'utils', 'von_mises', 'weibull', 'wishart']
```

```
print(dir(np.random))
```

```
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
↪'__name__', '__package__', '__spec__', '_mx_nd_np', 'beta', 'chisquare',
↪'choice', 'exponential', 'gamma', 'gumbel', 'logistic', 'lognormal',
↪'multinomial', 'multivariate_normal', 'normal', 'pareto', 'power', 'rand',
↪'randint', 'randn', 'rayleigh', 'shuffle', 'uniform', 'weibull']
```

```
print(dir(jax.random))
```

```
['KeyArray', 'PRNGKey', 'PRNGKeyArray', '_PRNGKeyArray', '__builtins__', '__
↪cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '_
↪_spec__', 'ball', 'bernoulli', 'beta', 'bits', 'categorical', 'cauchy',
↪'chisquare', 'choice', 'default_prng_impl', 'dirichlet', 'double_sided_
↪maxwell', 'exponential', 'f', 'fold_in', 'gamma', 'generalized_normal',
↪'geometric', 'gumbel', 'key', 'key_data', 'laplace', 'loggamma', 'logistic',
↪'maxwell', 'multivariate_normal', 'normal', 'orthogonal', 'pareto',
↪'permutation', 'poisson', 'rademacher', 'randint', 'random_gamma_p',
↪'rayleigh', 'rbg_key', 'shuffle', 'split', 't', 'threefry2x32_key',
```

(continues on next page)

(continued from previous page)

```
↪'threefry2x32_p', 'threefry_2x32', 'truncated_normal', 'typing', 'uniform',
↪'unsafe_rbg_key', 'wald', 'weibull_min']
```

```
print(dir(tf.random))
```

```
['Algorithm', 'Generator', '__builtins__', '__cached__', '__doc__', '__file__',
↪ '__loader__', '__name__', '__package__', '__path__', '__spec__', '_sys',
↪'all_candidate_sampler', 'categorical', 'create_rng_state', 'experimental',
↪'fixed_unigram_candidate_sampler', 'fold_in', 'gamma', 'get_global_generator
↪', 'learned_unigram_candidate_sampler', 'log_uniform_candidate_sampler',
↪'normal', 'poisson', 'set_global_generator', 'set_seed', 'shuffle', 'split',
↪'stateless_binomial', 'stateless_categorical', 'stateless_gamma', 'stateless_
↪normal', 'stateless_parameterized_truncated_normal', 'stateless_poisson',
↪'stateless_truncated_normal', 'stateless_uniform', 'truncated_normal',
↪'uniform', 'uniform_candidate_sampler']
```

一般に、__ で始まり __ で終わる関数（Python の特別なオブジェクト）や、単一の _ で始まる関数（通常は内部関数）は無視して構わない。残った関数名や属性名から判断すると、このモジュールは一様分布（uniform）、正規分布（normal）、多項分布（multinomial）からのサンプリングを含む、さまざまな乱数生成メソッドを提供していると推測できる。

5.7.2 特定の関数とクラス

特定の関数やクラスの使い方について詳しく知るには、help 関数を呼び出す。例として、テンソルの ones 関数の使い方を調べてみましょう。

```
help(torch.ones)
```

```
Help on built-in function ones in module torch:

ones(...)
    ones(*size, *, out=None, dtype=None, layout=torch.strided, device=None,
↪ requires_grad=False) -> Tensor

    Returns a tensor filled with the scalar value 1, with the shape defined
    by the variable argument size.

    Args:
        size (int...): a sequence of integers defining the shape of the␣
↪output tensor.
            Can be a variable number of arguments or a collection like a␣
↪list or tuple.

    Keyword arguments:
```

```
        out (Tensor, optional): the output tensor.
        dtype (torch.dtype, optional): the desired data type of returned␣
↪tensor.
            Default: if None, uses a global default (see torch.set_default_
↪tensor_type()).
        layout (torch.layout, optional): the desired layout of returned␣
↪Tensor.
            Default: torch.strided.
        device (torch.device, optional): the desired device of returned␣
↪tensor.
            Default: if None, uses the current device for the default tensor␣
↪type
            (see torch.set_default_tensor_type()). device will be the CPU
            for CPU tensor types and the current CUDA device for CUDA tensor␣
↪types.
        requires_grad (bool, optional): If autograd should record operations␣
↪on the
            returned tensor. Default: False.

    Example::

        >>> torch.ones(2, 3)
        tensor([[ 1.,  1.,  1.],
                [ 1.,  1.,  1.]])

        >>> torch.ones(5)
        tensor([ 1.,  1.,  1.,  1.,  1.])
```

```
help(np.ones)
```

```
Help on function ones in module mxnet.numpy:

ones(shape, dtype=<class 'numpy.float32'>, order='C', ctx=None)
    Return a new array of given shape and type, filled with ones.
    This function currently only supports storing multi-dimensional data
    in row-major (C-style).

    Parameters
    ----------
    shape : int or tuple of int
        The shape of the empty array.
    dtype : str or numpy.dtype, optional
        An optional value type. Default is numpy.float32. Note that this
        behavior is different from NumPy's ones function where float64
        is the default value, because float32 is considered as the default
```

```
        data type in deep learning.
    order : {'C'}, optional, default: 'C'
        How to store multi-dimensional data in memory, currently only row-major
        (C-style) is supported.
    ctx : Context, optional
        An optional device context (default is the current default context).
↪

    Returns
    -------
    out : ndarray
        Array of ones with the given shape, dtype, and ctx.

    Examples
    --------
    >>> np.ones(5)
    array([1., 1., 1., 1., 1.])

    >>> np.ones((5,), dtype=int)
    array([1, 1, 1, 1, 1], dtype=int64)

    >>> np.ones((2, 1))
    array([[1.],
           [1.]])

    >>> s = (2,2)
    >>> np.ones(s)
    array([[1., 1.],
           [1., 1.]])
```

```
help(jax.numpy.ones)
```

```
Help on function ones in module jax._src.numpy.lax_numpy:

ones(shape: Any, dtype: Union[Any, str, numpy.dtype, jax._src.SupportsDType,
↪ NoneType] = None) -> jax.Array
    Return a new array of given shape and type, filled with ones.

    LAX-backend implementation of numpy.ones().

    Original docstring below.

    Parameters
    ----------
    shape : int or sequence of ints
```

```
        Shape of the new array, e.g., (2, 3) or 2.
    dtype : data-type, optional
        The desired data-type for the array, e.g., numpy.int8.  Default␣
↪is
        numpy.float64.

    Returns
    -------
    out : ndarray
        Array of ones with the given shape, dtype, and order.
```

```
help(tf.ones)
```

```
Help on function ones in module tensorflow.python.ops.array_ops:

ones(shape, dtype=tf.float32, name=None)
    Creates a tensor with all elements set to one (1).

    See also tf.ones_like, tf.zeros, tf.fill, tf.eye.

    This operation returns a tensor of type dtype with shape shape and
    all elements set to one.

    >>> tf.ones([3, 4], tf.int32)
    <tf.Tensor: shape=(3, 4), dtype=int32, numpy=
    array([[1, 1, 1, 1],
           [1, 1, 1, 1],
           [1, 1, 1, 1]], dtype=int32)>

    Args:
      shape: A list of integers, a tuple of integers, or
        a 1-D Tensor of type int32.
      dtype: Optional DType of an element in the resulting Tensor. Default␣
↪is
        tf.float32.
      name: Optional string. A name for the operation.

    Returns:
      A Tensor with all elements set to one (1).
```

ドキュメントから、ones 関数は指定された形状をもつ新しいテンソルを作成し、すべての要素を 1 に設定することがわかる。可能な限り、解釈が正しいことを確認するために簡単なテストを実行するべきである。

```
torch.ones(4)
```

```
tensor([1., 1., 1., 1.])
```

```
np.ones(4)
```

```
[07:00:06] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
array([1., 1., 1., 1.])
```

```
jax.numpy.ones(4)
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
Array([1., 1., 1., 1.], dtype=float32)
```

```
tf.ones(4)
```

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([1., 1., 1., 1.],␣
↪dtype=float32)>
```

Jupyter ノートブックでは、? を使ってドキュメントを別ウィンドウに表示できる。たとえば、list? は help(list) とほぼ同じ内容を生成し、新しいブラウザウィンドウに表示す。さらに、list?? のように疑問符を 2 つ使うと、その関数を実装している Python コードも表示される。

公式ドキュメントには、この本の範囲を超える豊富な説明と実例が掲載されている。私たちは、単なる網羅的なリストを作るよりも、実際に問題を解決するために役立つ重要な使用例を重視して紹介していきる。また、ライブラリのソースコードを調べて、本番コードの高品質な実装例を見ることも勧める。そうすることで、より優れた科学者になるだけでなく、より優れたエンジニアにもなれるだろう。

6 回帰のための線形ニューラルネットワーク

ディープニューラルネットワークを考える前に、入力が出力に直接つながるような、層の浅いネットワークを実装しておくことは有益である。これにはいくつか重要な理由がある。第一に、複雑なアーキテクチャに気を取られる代わりに、出力層のパラメータ化、データの扱い、損失関数の指定、モデルの学習といった、ニューラルネットワーク学習の基本に集中できる。第二に、この種の浅いネットワークは線形モデルの集合そのものであり、線形回帰やソフトマックス回帰を含む、多くの古典的な統計的予測手法を包含している。これらの古典的な手法を理解することは極めて重要である。なぜなら、それらは多くの場面で広く使われており、より洗練されたアーキテクチャの利用を正当化するときには、しばしばベースラインとして用いる必要があるからである。この章では線形回帰に焦点を当て、次の章で分類のための線形ニューラルネットワークを構築することで、私たちのモデリング手法の幅を広げる。

6.1 線形回帰

回帰問題は、数値を予測したいときに現れる。代表的な例としては、価格（住宅、株式など）の予測、入院患者の滞在期間の予測、需要予測（小売売上）など、数多くある。すべての予測問題が古典的な回帰であるわけではない。後ほど、カテゴリ集合の中での所属を予測することを目的とする分類問題を紹介する。

例として、家の面積（平方フィート）と築年数（年）に基づいて、住宅価格（ドル）を推定したいとする。住宅価格を予測するモデルを構築するには、各住宅について売却価格、面積、築年数を含むデータを手に入れる必要がある。機械学習の用語では、このデータセットは 訓練データセットまたは 訓練集合と呼ばれ、各行（1 件の売買に対応するデータを含む）は 例（または データ例、インスタンス、サンプル）と呼ばれる。私たちが予測しようとしているもの（価格）は ラベル（または ターゲット）と呼ばれる。予測の基礎となる変数（築年数と面積）は 特徴量（または 共変量）と呼ばれる。

```
%matplotlib inline
from d2l import torch as d2l
import math
import torch
import numpy as np
import time
```

```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import np
import time
```

```
%matplotlib inline
from d2l import jax as d2l
from jax import numpy as jnp
import math
import time
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import math
import tensorflow as tf
import numpy as np
import time
```

6.1.1 基礎

線形回帰は、回帰問題に取り組むための標準的な手法の中で最も単純であり、かつ最も広く使われている。19 世紀初頭にさかのぼる線形回帰 (Gauss, 1809, Legendre, 1805) は、いくつかの単純な仮定から導かれる。まず、特徴量 $\mathbf{x}$ とターゲット y の関係がおおむね線形である、すなわち条件付き平均 $E[Y \mid X = \mathbf{x}]$ が特徴量 $\mathbf{x}$ の重み付き和として表せると仮定する。この設定では、観測ノイズのためにターゲット値が期待値からずれることは許される。次に、そのようなノイズがガウス分布に従う、よく振る舞うものだと仮定できる。通常、n をデータセット中の例の数を表すのに用いる。サンプルとターゲットの列挙には上付き添字を、座標の添字には下付き添字を用いる。より具体的には、$\mathbf{x}^{(i)}$ は i 番目のサンプルを表し、$x_j^{(i)}$ はその j 番目の座標を表す。

モデル

あらゆる解法の中心には、特徴量をターゲットの推定値へと変換する方法を記述するモデルがある。線形性の仮定とは、ターゲット（価格）の期待値が特徴量（面積と築年数）の重み付き和として表せることを意味する。

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b. \tag{6.1.1}$$

ここで w_{area} と w_{age} は 重みと呼ばれ、b は バイアス（または オフセット、切片）と呼ばれる。重みは、各特徴量が予測に与える影響を決定する。バイアスは、すべての特徴量が 0 のときの推定値を決定する。新築住宅で面積が厳密に 0

のものは決して見られないが、それでもバイアスが必要なのは、特徴量のすべての線形関数を表現できるようにするためです（原点を通る直線に限定されないようにするためです）。厳密には、(6.1.1) は入力特徴量の アフィン変換であり、特徴量の 線形変換（重み付き和）と、加えられたバイアスによる 平行移動から成る。データセットが与えられたとき、私たちの目標は重み $\mathbf{w}$ とバイアス b を選び、平均的に見て、モデルの予測がデータ中で観測された真の価格にできるだけよく一致するようにすることである。

特徴量が少数のデータセットに注目することが一般的な分野では、(6.1.1) のようにモデルを明示的に長い形で表すことがよくある。機械学習では、通常は高次元データセットを扱うため、コンパクトな線形代数の記法を用いる方が便利である。入力が d 個の特徴量からなるとき、それぞれに（1 から d までの）添字を割り当て、予測 $\hat{y}$ を（一般に「ハット」記号は推定値を表す）次のように表せる。

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b. \tag{6.1.2}$$

すべての特徴量をベクトル $\mathbf{x} \in \mathbb{R}^d$ に、すべての重みをベクトル $\mathbf{w} \in \mathbb{R}^d$ にまとめると、モデルは $\mathbf{w}$ と $\mathbf{x}$ の内積を用いて簡潔に表せる。

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b. \tag{6.1.3}$$

(6.1.3) では、ベクトル $\mathbf{x}$ は 1 つの例の特徴量に対応する。データセット全体が n 個の例からなるときには、設計行列 $\mathbf{X} \in \mathbb{R}^{n \times d}$ を用いて特徴量を表すと便利なことがよくある。ここで、$\mathbf{X}$ は各例に対して 1 行、各特徴量に対して 1 列を持つ。特徴量の集合 $\mathbf{X}$ に対して、予測 $\hat{\mathbf{y}} \in \mathbb{R}^n$ は行列–ベクトル積で表せる。

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b, \tag{6.1.4}$$

ここでは、和を取る際にブロードキャスト（Section 5.1.4）が適用される。訓練データセットの特徴量 $\mathbf{X}$ と対応する（既知の）ラベル $\mathbf{y}$ が与えられたとき、線形回帰の目標は、$\mathbf{X}$ と同じ分布からサンプリングされた新しいデータ例の特徴量が与えられたときに、その新しい例のラベルを（期待値として）最小誤差で予測できるような重みベクトル $\mathbf{w}$ とバイアス項 b を見つけることである。

たとえ、$\mathbf{x}$ が与えられたときの y を予測する最良のモデルが線形であると信じていたとしても、現実世界の n 個の例からなるデータセットですべての $1 \leq i \leq n$ に対して $y^{(i)} = \mathbf{w}^\top \mathbf{x}^{(i)} + b$ が厳密に成り立つとは期待しない。たとえば、$\mathbf{X}$ と $\mathbf{y}$ を観測するためにどのような機器を使っても、わずかな測定誤差があるかもしれない。したがって、基礎となる関係が線形であると確信している場合でも、そのような誤差を考慮するためにノイズ項を組み込みる。

最良の パラメータ（または モデルパラメータ）$\mathbf{w}$ と b を探し始める前に、さらに 2 つ必要である。(i) 与えられたモデルの品質を測る尺度、(ii) モデルの品質を改善するためにモデルを更新する手順である。

損失関数

当然ながら、モデルをデータに適合させるには、適合度（あるいは同値に 不適合度）の尺度について合意する必要がある。損失関数は、ターゲットの 真の値と 予測値の間の距離を定量化する。損失は通常、非負の数であり、小さいほど良く、完全な予測では損失 0 になる。回帰問題で最も一般的な損失関数は二乗誤差である。例 i に対する予測が $\hat{y}^{(i)}$、対応する真のラベルが $y^{(i)}$ のとき、二乗誤差は次のように与えられる。

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{6.1.5}$$

定数 $\frac{1}{2}$ は本質的には違いを生まないが、損失の微分を取るときに打ち消し合うので、記法上便利である。訓練データセットは与えられており、したがって私たちの制御外にあるため、経験誤差はモデルパラメータの関数にすぎない。Fig. 6.1.1 では、1 次元入力の問題における線形回帰モデルの適合を可視化している。

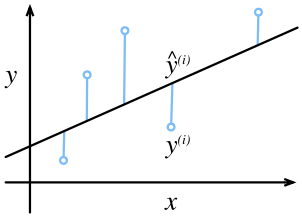


Fig. 6.1.1 1 次元データに線形回帰モデルを適合させる。

推定値 $\hat{y}^{(i)}$ とターゲット $y^{(i)}$ の差が大きいと、その二次形式のために損失への寄与はさらに大きくなる（この二次性は諸刃の剣である。大きな誤差を避けるようモデルを促す一方で、異常データに対して過度に敏感になることもある）。n 個の例からなるデータセット全体に対するモデルの品質を測るには、訓練集合上の損失を単純に平均（あるいは同値に総和）すればよいである。

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{6.1.6}$$

モデルを訓練するとき、私たちはすべての訓練例にわたる総損失を最小にするパラメータ $(\mathbf{w}^*, b^*)$ を求める。

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}}\ L(\mathbf{w}, b). \tag{6.1.7}$$

解析解

私たちが扱うほとんどのモデルとは異なり、線形回帰は驚くほど簡単な最適化問題を与えてくれる。特に、単純な公式を適用することで、（訓練データ上で評価した）最適パラメータを解析的に求めることができる。まず、設計行列にすべて 1 からなる列を追加することで、バイアス b をパラメータ $\mathbf{w}$ に組み込める。すると、予測問題は $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$ を最小化することになる。設計行列 $\mathbf{X}$ がフ

ルランク（どの特徴量も他の特徴量の線形従属でない）である限り、損失面にはただ 1 つの臨界点しかなく、それは定義域全体にわたる損失の最小値に対応する。損失を $\mathbf{w}$ で微分して 0 に等置すると、次が得られる。

$$\partial_{\mathbf{w}}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 = 2\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \text{ and hence } \mathbf{X}^\top\mathbf{y} = \mathbf{X}^\top\mathbf{X}\mathbf{w}. \tag{6.1.8}$$

$\mathbf{w}$ について解けば、最適化問題の最適解が得られる。この解は

$$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y} \tag{6.1.9}$$

が一意であるのは、行列 $\mathbf{X}^\top\mathbf{X}$ が可逆な場合、すなわち設計行列の列が線形独立である場合に限られます (Golub and Van Loan, 1996)。

線形回帰のような単純な問題では解析解が得られることがあるが、そのような幸運に慣れてはいけない。解析解は美しい数学的解析を可能にするが、解析解を要求することは非常に制約が強く、深層学習のほとんどすべての魅力的な側面を排除してしまいる。

ミニバッチ確率的勾配降下法

幸いなことに、モデルを解析的に解けない場合でも、実際にはしばしば効果的にモデルを訓練できる。さらに、多くのタスクでは、そのような最適化の難しいモデルの方がはるかに優れているため、訓練方法を見つける手間は十分に見合うものになる。

ほぼすべての深層学習モデルを最適化するための鍵となる技法であり、本書全体を通して用いることになるのが、パラメータを少しずつ損失関数を下げる方向に更新することで誤差を反復的に減らす方法である。このアルゴリズムは 勾配降下法と呼ばれる。

勾配降下法の最も素朴な適用は、データセット中のすべての例で計算された損失の平均である損失関数の微分を取ることである。実際には、これは非常に遅くなることがある。更新が非常に強力であっても、1 回更新する前にデータセット全体を 1 周しなければならないからです (Liu and Nocedal, 1989)。さらに悪いことに、訓練データに冗長性が多い場合、フル更新の利点は限られる。

もう一方の極端は、一度に 1 つの例だけを考え、1 つの観測に基づいて更新することである。その結果得られるアルゴリズムである 確率的勾配降下法（SGD）は、大規模データセットに対しても有効な戦略になりえます (Bottou, 2010)。残念ながら、SGD には計算上および統計上の欠点がある。1 つの問題は、プロセッサが主記憶からプロセッサキャッシュへデータを移動するよりも、数値の乗算や加算をはるかに高速に行えるという事実から生じる。行列–ベクトル積を実行する方が、対応する数のベクトル–ベクトル演算を行うよりも、最大で 1 桁程度効率的である。これは、1 サンプルずつ処理するのにフルバッチよりもはるかに時間がかかりうることを意味する。2 つ目の問題は、バッチ正規化（Section 11.5 で説明する）のような一部の層は、一度に複数の観測にアクセスできる場合にのみうまく機能することである。

これら 2 つの問題に対する解決策は、中間的な戦略を選ぶことである。フル

バッチでも 1 サンプルずつでもなく、観測の ミニバッチを取ります (Li *et al.*, 2014)。このミニバッチのサイズの具体的な選択は、メモリ量、アクセラレータの数、層の選択、データセット全体のサイズなど、多くの要因に依存する。それでも、32 から 256 の間の数、できれば 2 の大きな冪の倍数がよい出発点である。これが ミニバッチ確率的勾配降下法につながりる。

最も基本的な形では、各反復 t において、まず固定個数 $|\mathcal{B}|$ の訓練例からなるミニバッチ $\mathcal{B}_t$ をランダムにサンプリングする。次に、ミニバッチ上の平均損失をモデルパラメータに関して微分（勾配計算）する。最後に、その勾配に 学習率と呼ばれるあらかじめ定めた小さな正の値 η を掛け、得られた項を現在のパラメータ値から引きる。更新は次のように表せる。

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b). \tag{6.1.10}$$

要するに、ミニバッチ SGD は次のように進みる。(i) 通常はランダムに、モデルパラメータの値を初期化する。(ii) データからランダムなミニバッチを反復的にサンプリングし、負の勾配の方向にパラメータを更新する。二次損失とアフィン変換に対しては、これは閉形式で展開できる。

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) &&= \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right) \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_b l^{(i)}(\mathbf{w}, b) &&= b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right). \end{aligned} \tag{6.1.11}$$

ミニバッチ $\mathcal{B}$ を選ぶので、そのサイズ $|\mathcal{B}|$ で正規化する必要がある。しばしば、ミニバッチサイズと学習率はユーザが定義する。訓練ループの中で更新されないこのような調整可能なパラメータは ハイパーパラメータと呼ばれる。それらはベイズ最適化などのいくつかの手法で自動的に調整できる (Frazier, 2018)。最終的に、解の品質は通常、別の 検証データセット（または 検証集合）で評価される。

あらかじめ定めた回数だけ反復して訓練した後（あるいは別の停止条件が満たされた後）、推定されたモデルパラメータ $\hat{\mathbf{w}}, \hat{b}$ を記録する。たとえ関数が本当に線形でノイズがないとしても、これらのパラメータは損失の厳密な最小化解にはならず、ましてや決定論的でもない。アルゴリズムは最小化解に向かってゆっくり収束するが、有限回のステップでそれらを厳密に見つけることは通常ない。さらに、パラメータ更新に用いるミニバッチ $\mathcal{B}$ はランダムに選ばれる。これにより決定論性は失われる。

線形回帰は、たまたま大域的最小値を持つ学習問題です（$\mathbf{X}$ がフルランクであるとき、あるいは同値に $\mathbf{X}^\top \mathbf{X}$ が可逆であるとき）。しかし、深層ネットワークの損失面には多くの鞍点と最小値がある。幸いなことに、私たちは通常、厳密なパラメータ集合を見つけることには関心がなく、正確な予測（したがって低い損失）を与える何らかのパラメータ集合が得られれば十分である。実際には、深層学習の実践者は訓練集合上で損失を最小化するパラメータを見つけるのに苦労することはめったにない (Frankle and Carbin, 2018, Izmailov *et al.*, 2018)。より困難なのは、これまで見たことのないデータに対して正確な予測を与える

パラメータを見つけることである。この課題は 汎化と呼ばれる。これらの話題には本書を通して何度も立ち返りる。

予測

モデル $\hat{\mathbf{w}}^\top \mathbf{x} + \hat{b}$ が与えられれば、新しい例に対して 予測を行える。たとえば、面積 x_1 と築年数 x_2 が与えられたときに、これまで見たことのない住宅の売却価格を予測できる。深層学習の実践者は予測段階を 推論と呼ぶようになっているが、これは少し不正確である。推論は一般に、証拠に基づいて到達したあらゆる結論を指し、パラメータの値と、見えないインスタンスのラベルの可能性の両方を含む。むしろ統計学の文献では、推論はパラメータ推論を指すことの方が多く、この用語の多義性は、深層学習の実践者が統計学者と話すときに不要な混乱を生みる。以下では、可能な限り 予測を用いる。

6.1.2 高速化のためのベクトル化

モデルを訓練するとき、通常は例のミニバッチ全体を同時に処理したいと考える。これを効率よく行うには、計算をベクトル化し、Python で高価な for ループを書くのではなく、高速な線形代数ライブラリを活用する必要がある。

これがなぜそれほど重要なのかを見るために、ベクトルを加算する 2 つの方法を考えてみよう。まず、すべて 1 からなる 10,000 次元ベクトルを 2 つ用意する。1 つ目の方法では、Python の for ループでベクトルを走査する。2 つ目では、+ を 1 回呼び出すだけに頼りる。

```
n = 10000
a = d2l.ones(n)
b = d2l.ones(n)
```

```
[06:59:48] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

これで処理時間をベンチマークできる。まず、for ループを使って、1 座標ずつ加算する。

```
c = d2l.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

```
'0.12482 sec'
```

```
c = d2l.zeros(n)
t = time.time()
```

(continues on next page)

(continued from previous page)

```
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

```
'4.90001 sec'
```

```
# JAX arrays are immutable, meaning that once created their contents
# cannot be changed. For updating individual elements, JAX provides
# an indexed update syntax that returns an updated copy
c = d2l.zeros(n)
t = time.time()
for i in range(n):
    c = c.at[i].set(a[i] + b[i])
f'{time.time() - t:.5f} sec'
```

```
'24.00628 sec'
```

```
c = tf.Variable(d2l.zeros(n))
t = time.time()
for i in range(n):
    c[i].assign(a[i] + b[i])
f'{time.time() - t:.5f} sec'
```

```
'12.79235 sec'
```

あるいは、再ロードされた + 演算子を使って要素ごとの和を計算する。

```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

```
'0.00029 sec'
```

2 つ目の方法は 1 つ目よりも劇的に高速である。コードをベクトル化すると、しばしば桁違いの高速化が得られる。さらに、数学の多くをライブラリ側に任せられるので、自分で計算を書く量が減り、誤りの可能性が下がり、コードの移植性も高まりる。

6.1.3 正規分布と二乗損失

ここまで、二乗損失目的関数についてかなり実用的な動機づけを与えてきた。すなわち、基礎となるパターンが本当に線形であるなら、最適パラメータは条件付き期待値 $E[Y \mid X]$ を返し、またこの損失は外れ値に大きな罰則を与える。

さらに、ノイズの分布について確率的な仮定を置くことで、二乗損失目的関数に対してより形式的な動機づけも与えられる。

線形回帰は 19 世紀の変わり目に発明された。ガウスとルジャンドルのどちらが最初にこのアイデアを思いついたかは長く議論されてきたが、正規分布（ガウス分布とも呼ばれます）を発見したのもガウスでした。正規分布と二乗損失を伴う線形回帰は、単に同じ祖先を持つという以上の深い関係を共有していることがわかる。

まず、平均 μ、分散 σ^2（標準偏差 σ）の正規分布は次で与えられる。

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right). \tag{6.1.12}$$

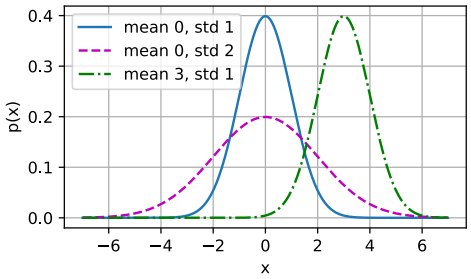
以下では正規分布を計算する関数を定義する。

```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    if tab.selected('jax'):
        return p * jnp.exp(-0.5 * (x - mu)**2 / sigma**2)
    if tab.selected('pytorch', 'mxnet', 'tensorflow'):
        return p * np.exp(-0.5 * (x - mu)**2 / sigma**2)
```

これで正規分布を可視化できる。

```
if tab.selected('jax'):
    # Use JAX NumPy for visualization
    x = jnp.arange(-7, 7, 0.01)
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    # Use NumPy again for visualization
    x = np.arange(-7, 7, 0.01)

# Mean and standard deviation pairs
params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
         ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```

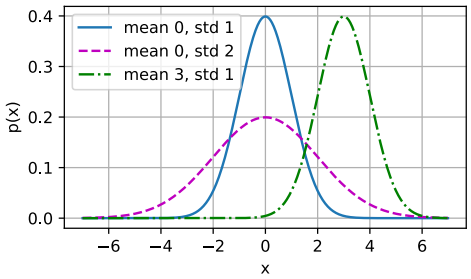



```
# Use NumPy again for visualization
x = np.arange(-7, 7, 0.01)
```

(continues on next page)

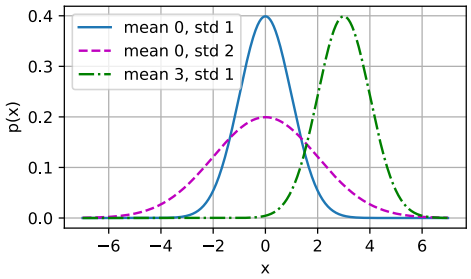
(continued from previous page)

```
# Mean and standard deviation pairs
params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x.asnumpy(), [normal(x, mu, sigma).asnumpy() for mu, sigma in params],
↪ xlabel='x',
         ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```




```
if tab.selected('jax'):
    # Use JAX NumPy for visualization
    x = jnp.arange(-7, 7, 0.01)
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    # Use NumPy again for visualization
    x = np.arange(-7, 7, 0.01)

# Mean and standard deviation pairs
params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
         ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```

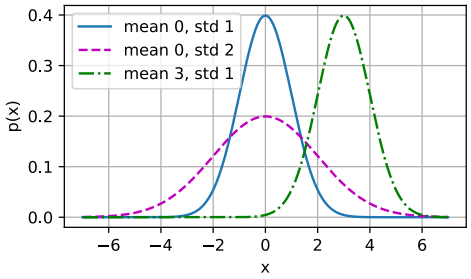



```
if tab.selected('jax'):
    # Use JAX NumPy for visualization
    x = jnp.arange(-7, 7, 0.01)
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    # Use NumPy again for visualization
    x = np.arange(-7, 7, 0.01)
```

(continues on next page)

(continued from previous page)

```
# Mean and standard deviation pairs
params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
         ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```



平均を変えると x 軸に沿った平行移動に対応し、分散を大きくすると分布は広がり、ピークは低くなる。

二乗損失を伴う線形回帰を動機づける 1 つの方法は、観測がノイズ付き測定から生じると仮定し、そのノイズ ϵ が正規分布 $\mathcal{N}(0, \sigma^2)$ に従うとすることである。

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{6.1.13}$$

したがって、与えられた $\mathbf{x}$ に対して特定の y が観測される 尤度は次のように書ける。

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \tag{6.1.14}$$

このように、尤度は因数分解される。最尤原理によれば、パラメータ $\mathbf{w}$ と b の最良の値は、データセット全体の 尤度を最大化するものである。

$$P(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} p(y^{(i)} \mid \mathbf{x}^{(i)}). \tag{6.1.15}$$

この等式は、すべての組 $(\mathbf{x}^{(i)}, y^{(i)})$ が互いに独立に抽出されたために成り立ちる。最尤原理に従って選ばれる推定量は 最尤推定量と呼ばれる。多くの指数関数の積を最大化するのは難しそうに見えるが、目的関数を変えずに、尤度の対数を最大化することで大幅に簡単にできる。歴史的な理由から、最適化は最大化よりも最小化として表現されることが多いである。そこで、何も変えずに、負の対数尤度を 最小化できる。これは次のように表せる。

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \tag{6.1.16}$$

σ が固定であると仮定すれば、第 1 項は $\mathbf{w}$ や b に依存しないので無視できる。

第 2 項は、前に導入した二乗誤差損失と同じであり、乗法定数 $\frac{1}{\sigma^2}$ が付いているだけである。幸いなことに、解は
sigma にも依存しない。したがって、平均二乗誤差を最小化することは、加法的ガウスノイズを仮定した線形モデルの最尤推定と等価である。

6.1.4 ニューラルネットワークとしての線形回帰

線形モデルは、本書でこれから紹介する複雑なネットワークを表現するには十分に豊かではないが、（人工）ニューラルネットワークは、すべての特徴量が入力ニューロンで表され、それらがすべて出力に直接接続されたネットワークとして線形モデルを包含できるほど豊かである。

Fig. 6.1.2 は線形回帰をニューラルネットワークとして描いている。この図は、各入力が出力にどのように接続されているかといった接続パターンを強調しているが、重みやバイアスが取る具体的な値は示していない。

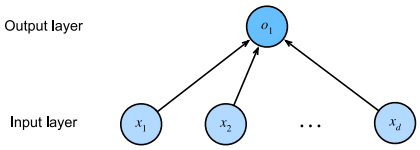


Fig. 6.1.2 線形回帰は 1 層ニューラルネットワークである。

入力は $x_1, \ldots, x_d$ である。d を入力層における 入力数または 特徴次元と呼ぶ。ネットワークの出力は o_1 である。私たちは単一の数値を予測したいだけなので、出力ニューロンは 1 つしかない。入力値はすべて 与えられたものであることに注意されたい。計算されるニューロンは 1 つだけである。要するに、線形回帰は 1 層の全結合ニューラルネットワークとみなせる。後の章では、はるかに多くの層を持つネットワークに出会いる。

生物学

線形回帰は計算神経科学よりも前に登場したため、線形回帰をニューラルネットワークの観点で説明するのは時代錯誤に見えるかもしれない。それでも、サイバネティクス研究者で神経生理学者のウォーレン・マカロックとウォルター・ピッツが人工ニューロンのモデルを開発し始めたとき、それは自然な出発点でした。Fig. 6.1.3 に示す生物学的ニューロンの漫画的な図を考えてみよう。そこには、樹状突起（入力端子）、細胞核（CPU）、軸索（出力線）、および軸索終末（出力端子）があり、シナプスを介して他のニューロンとの接続を可能にしている。

他のニューロン（あるいは環境センサー）から到着する情報 x_i は樹状突起で受け取られる。特に、その情報は シナプス重み w_i によって重み付けされ、積 $x_i w_i$ による活性化や抑制など、入力の効果を決定する。複数のソースから到着した重み付き入力は細胞核で重み付き和 $y = \sum_i x_i w_i + b$ として集約され、場合によっては関数 $\sigma(y)$ による非線形な後処理を受ける。この情報はその後、軸索を通って軸索終末へ送られ、そこで目的地（たとえば筋肉のようなアクチュ

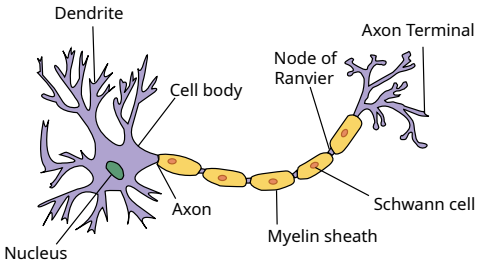


Fig. 6.1.3 実際のニューロン（出典: 米国国立がん研究所 Surveillance, Epidemiology and End Results (SEER) Program の“Anatomy and Physiology”）。

エータ）に到達するか、あるいは樹状突起を介して別のニューロンに入力される。

もちろん、正しい接続性と学習アルゴリズムを備えていれば、このような多数の単位を組み合わせることで、単一のニューロンだけでは表現できないはるかに興味深く複雑な振る舞いを生み出せるという高レベルの考え方は、実際の生物学的神経系の研究から生まれたものである。同時に、今日の深層学習研究の大部分はより広い源泉から着想を得ている。Russell and Norvig (2016) を引く。彼らは、飛行機が鳥に 着想を得たものであっても、鳥類学がここ数世紀の航空工学革新の主たる原動力ではなかったと指摘した。同様に、今日の深層学習における着想は、数学、言語学、心理学、統計学、計算機科学、そして他の多くの分野から、同程度かそれ以上に得られている。

6.1.5 要約

この節では、伝統的な線形回帰を導入した。そこでは、線形関数のパラメータを訓練集合上の二乗損失を最小化するように選ぶ。また、この目的関数の選択を、いくつかの実用的な考慮と、線形性およびガウスノイズの仮定の下での線形回帰の最尤推定としての解釈の両方から動機づけた。計算上の考慮と統計との関係の両方を議論した後、このような線形モデルを、入力が出力に直接配線された単純なニューラルネットワークとしてどのように表せるかを示した。まもなく線形モデルからは完全に先へ進みるが、それらは、すべてのモデルに必要なほとんどの構成要素　パラメトリックな形、微分可能な目的関数、ミニバッチ確率的勾配降下法による最適化、そして最終的には、これまで見たことのないデータでの評価　を導入するのに十分である。

6.1.6 演習

1. いくつかのデータ $x_1, \ldots, x_n \in \mathbb{R}$ があると仮定する。私たちの目標は、$\sum_i (x_i - b)^2$ が最小になる定数 b を見つけることである。

    1. b の最適値の解析解を求めなさい。

    2. この問題とその解は正規分布とどのように関係しているか？

3. 損失を $\sum_i (x_i - b)^2$ から $\sum_i |x_i - b|$ に変えたらどうなるか？b の最適解を見つけられますか？

2. $\mathbf{x}^\top \mathbf{w} + b$ で表せるアフィン関数が、$(\mathbf{x}, 1)$ 上の線形関数と等価であることを証明しなさい。

3. $\mathbf{x}$ の二次関数、すなわち $f(\mathbf{x}) = b + \sum_i w_i x_i + \sum_{j \leq i} w_{ij} x_i x_j$ を求めたいとする。これを深層ネットワークでどのように定式化するか？

4. 線形回帰問題が解ける条件の 1 つは、設計行列 $mathbf X^{top mathbf X}$ がフルランクであることだったことを思い出しよう。

   1. そうでない場合はどうなるか？

   2. どうすれば修正できるか？$\mathbf{X}$ のすべての要素に、座標ごとに独立なガウスノイズを少量加えるとどうなるか？

   3. この場合、設計行列 $\mathbf{X}^\top \mathbf{X}$ の期待値はいくらですか？

   4. $\mathbf{X}^\top \mathbf{X}$ がフルランクでないとき、確率的勾配降下法では何が起こりますか？

5. 加法ノイズ ϵ を支配するノイズモデルが指数分布であると仮定する。すなわち、$p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$ である。

   1. このモデルの下でのデータの負の対数尤度 $-\log P(\mathbf{y} \mid \mathbf{X})$ を書き下しなさい。

   2. 閉形式解を見つけられますか？

   3. この問題を解くためのミニバッチ確率的勾配降下法アルゴリズムを提案しなさい。何がうまくいかなくなる可能性があるか（ヒント: パラメータを更新し続けると、停留点の近くで何が起こりますか）？それを修正できるか？

6. 2 つの線形層を合成して 2 層のニューラルネットワークを設計したいとする。すなわち、最初の層の出力が 2 番目の層の入力になる。なぜこのような素朴な合成はうまくいかないのだろうか？

7. 現実的な住宅価格や株価の推定に回帰を使いたいとすると、何が起こりますか？

   1. 加法的ガウスノイズの仮定が適切でないことを示しなさい。ヒント: 価格が負になることはあるか？変動はどうだろうか？

   2. 価格の対数に回帰する方が、つまり $y = \log \text{price}$ とする方が、なぜずっと良いのだろうか？

   3. ペニー株、すなわち非常に低価格の株を扱うとき、何に注意する必要があるか？ヒント: すべての可能な価格で取引できるか？なぜ安い株の方がこの問題は大きいのだろうか？詳しくは、オプション価格付けの有名な Black–Scholes モデルを復習せよ (Black and Scholes, 1973)。

8. 食料品店で売れたリンゴの 個数を推定するために回帰を使いたいとする。

   1. ガウス加法ノイズモデルの問題点は何ですか？ヒント: 売っているのは石油ではなくリンゴである。

   2. ポアソン分布[52] は個数に関する分布を表す。これは $p(k \mid \lambda) = \lambda^k e^{-\lambda}/k!$ で与えられる。ここで λ は率関数であり、k は観測される事象数である。λ が個数 k の期待値であることを証明しなさい。

   3. ポアソン分布に対応する損失関数を設計しなさい。

   4. 代わりに $\log \lambda$ を推定するための損失関数を設計しなさい。

6.2 実装のためのオブジェクト指向設計

線形回帰の導入では、データ、モデル、損失関数、最適化アルゴリズムを含むさまざまな構成要素を順に見てきた。実際、線形回帰は機械学習モデルの中でも最も単純なものの一つである。しかし、その学習には、この本の他のモデルでも必要となる多くの同じ構成要素が使われる。したがって、実装の詳細に入る前に、ここで使ういくつかの API を設計しておく価値がある。深層学習の構成要素をオブジェクトとして扱えば、それらのオブジェクトと相互作用を定義するクラスを作ることから始められる。この実装のためのオブジェクト指向設計は、説明を大幅に整理するだけでなく、実際のプロジェクトでも有用にお使いいただけるだろう。

PyTorch Lightning[53] のようなオープンソースライブラリに着想を得て、大まかには次の 3 つのクラスを用意したいと考える。(i) `Module` はモデル、損失、最適化手法を含む。(ii) `DataModule` は学習用と検証用のデータローダーを提供する。(iii) これら 2 つのクラスは `Trainer` クラスによって統合され、さまざまなハードウェアプラットフォーム上でモデルを学習できるようにする。この本のコードの大部分は `Module` と `DataModule` を拡張したものである。`Trainer` クラスに触れるのは、GPU、CPU、並列学習、最適化アルゴリズムを扱うときだけである。

```
import time
import numpy as np
from d2l import torch as d2l
import torch
from torch import nn
```

```
import time
import numpy as np
from d2l import mxnet as d2l
from mxnet.gluon import nn
```

```
from dataclasses import field
from d2l import jax as d2l
from flax import linen as nn
from flax.training import train_state
from jax import numpy as jnp
import numpy as np
import jax
import time
from typing import Any
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
import time
import numpy as np
from d2l import tensorflow as d2l
import tensorflow as tf
```

6.2.1 ユーティリティ

Jupyter ノートブックでオブジェクト指向プログラミングを簡単にするために、いくつかのユーティリティが必要である。課題の一つは、クラス定義がかなり長いコードブロックになりがちなことである。ノートブックの読みやすさを保つには、説明を挟みながら短いコード片を並べる必要があるが、これは Python ライブラリで一般的なプログラミングスタイルとは相性がよくない。最初のユーティリティ関数は、クラスが作成された後に、その関数をクラスのメソッドとして登録できるようにする。実際、クラスのインスタンスを作成した後であっても可能である。これにより、クラスの実装を複数のコードブロックに分割できる。

```
def add_to_class(Class):  #@save
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

使い方を簡単に見てみよう。do というメソッドを持つクラス A を実装したいとする。A と do のコードを同じコードブロックに書く代わりに、まずクラス A を宣言してインスタンス a を作成できる。

```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

次に、通常どおり do メソッドを定義するが、A クラスのスコープ内では定義し

ない。その代わり、このメソッドを add_to_class でデコレートし、引数としてクラス A を渡す。こうすることで、このメソッドは、あたかも A の定義の一部として含まれていたかのように、A のメンバー変数へアクセスできる。インスタンス a に対して呼び出すとどうなるか見てみよう。

```
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a.do()
```

```
Class attribute "b" is 1
```

2 つ目のユーティリティは、クラスの __init__ メソッドのすべての引数をクラス属性として保存するクラスである。これにより、追加のコードを書かずにコンストラクタの呼び出しシグネチャを暗黙的に拡張できる。

```
class HyperParameters:  #@save
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

その実装は Section 26.7 に回す。使うには、HyperParameters を継承し、__init__ メソッド内で save_hyperparameters を呼び出すクラスを定義する。

```
# Call the fully implemented HyperParameters class saved in d2l
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        self.save_hyperparameters(ignore=['c'])
        print('self.a =', self.a, 'self.b =', self.b)
        print('There is no self.c =', not hasattr(self, 'c'))

b = B(a=1, b=2, c=3)
```

```
self.a = 1 self.b = 2
There is no self.c = True
```

最後のユーティリティは、実験の進行状況をその場で対話的に描画できるようにするものである。はるかに強力で複雑な TensorBoard[54] に敬意を表して、これを ProgressBoard と名付ける。実装は Section 26.7 に回す。ここでは、まず動作だけ見てみよう。

[54]

draw メソッドは、図中に点 (x, y) を描画し、label を凡例に指定する。オプションの every_n は、図に $1/n$ 個の点だけを表示することで線を滑らかにする。それらの値は、元の図における n 個の近傍点から平均される。

```
class ProgressBoard(d2l.HyperParameters):  #@save
```

(continues on next page)

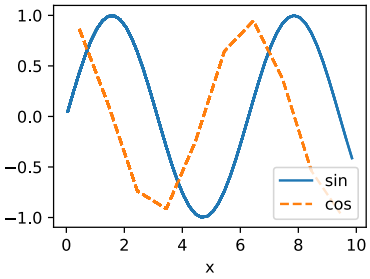
(continued from previous page)

```
    """The board that plots data points in animation."""
    def __init__(self, xlabel=None, ylabel=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 ls=['-', '--', '-.', ':'], colors=['C0', 'C1', 'C2', 'C3'],
                 fig=None, axes=None, figsize=(3.5, 2.5), display=True):
        self.save_hyperparameters()

    def draw(self, x, y, label, every_n=1):
        raise NotImplemented
```

次の例では、異なる滑らかさで sin と cos を描画する。このコードブロックを実行すると、線がアニメーションのように伸びていくのが見えるはずである。

```
board = d2l.ProgressBoard('x')
for x in np.arange(0, 10, 0.1):
    board.draw(x, np.sin(x), 'sin', every_n=2)
    board.draw(x, np.cos(x), 'cos', every_n=10)
```



6.2.2 モデル

Module クラスは、これから実装するすべてのモデルの基底クラスである。少なくとも 3 つのメソッドが必要である。1 つ目の __init__ は学習可能なパラメータを保存し、training_step メソッドはデータバッチを受け取って損失値を返す。最後に、configure_optimizers は学習可能なパラメータを更新するために使う最適化手法、またはそのリストを返す。必要に応じて、評価指標を報告するための validation_step を定義できる。出力の計算コードを別の forward メソッドに分けておくと、再利用しやすくなることがある。

```
class Module(d2l.nn_Module, d2l.HyperParameters):  #@save
    """The base class of models."""
    def __init__(self, plot_train_per_epoch=2, plot_valid_per_epoch=1):
        super().__init__()
        self.save_hyperparameters()
        self.board = ProgressBoard()

    def loss(self, y_hat, y):
```

(continues on next page)

(continued from previous page)

```
        raise NotImplementedError

    def forward(self, X):
        assert hasattr(self, 'net'), 'Neural network is defined'
        return self.net(X)

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
        self.board.draw(x, d2l.numpy(d2l.to(value, d2l.cpu())),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))

    def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=True)
        return l

    def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=False)

    def configure_optimizers(self):
        raise NotImplementedError
```

```
class Module(d2l.nn_Module, d2l.HyperParameters):  #@save
    """The base class of models."""
    if tab.selected('mxnet', 'tensorflow'):
        def __init__(self, plot_train_per_epoch=2, plot_valid_per_epoch=1):
            super().__init__()
            self.save_hyperparameters()
            self.board = ProgressBoard()
        if tab.selected('tensorflow'):
            self.training = None

    if tab.selected('jax'):
        # No need for save_hyperparam when using Python dataclass
        plot_train_per_epoch: int = field(default=2, init=False)
        plot_valid_per_epoch: int = field(default=1, init=False)
        # Use default_factory to make sure new plots are generated on each run
        board: ProgressBoard = field(default_factory=lambda: ProgressBoard(),
                                     init=False)

    def loss(self, y_hat, y):
```

(continues on next page)

(continued from previous page)

```
        raise NotImplementedError

    if tab.selected('mxnet', 'tensorflow'):
        def forward(self, X):
            assert hasattr(self, 'net'), 'Neural network is defined'
            return self.net(X)

    if tab.selected('tensorflow'):
        def call(self, X, *args, **kwargs):
            if kwargs and "training" in kwargs:
                self.training = kwargs['training']
            return self.forward(X, *args)

    if tab.selected('jax'):
        # JAX & Flax do not have a forward-method-like syntax. Flax uses setup
        # and built-in __call__ magic methods for forward pass. Adding here
        # for consistency
        def forward(self, X, *args, **kwargs):
            assert hasattr(self, 'net'), 'Neural network is defined'
            return self.net(X, *args, **kwargs)

        def __call__(self, X, *args, **kwargs):
            return self.forward(X, *args, **kwargs)

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
        if tab.selected('mxnet', 'tensorflow'):
            self.board.draw(x, d2l.numpy(value), (
                'train_' if train else 'val_') + key, every_n=int(n))
        if tab.selected('jax'):
            self.board.draw(x, d2l.to(value, d2l.cpu()),
                            ('train_' if train else 'val_') + key,
                            every_n=int(n))

    if tab.selected('mxnet', 'tensorflow'):
        def training_step(self, batch):
            l = self.loss(self(*batch[:-1]), batch[-1])
            self.plot('loss', l, train=True)
            return l

        def validation_step(self, batch):
            l = self.loss(self(*batch[:-1]), batch[-1])
            self.plot('loss', l, train=False)
```

(continues on next page)

(continued from previous page)

```
    if tab.selected('jax'):
        def training_step(self, params, batch, state):
            l, grads = jax.value_and_grad(self.loss)(params, batch[:-1],
                                                     batch[-1], state)
            self.plot("loss", l, train=True)
            return l, grads

        def validation_step(self, params, batch, state):
            l = self.loss(params, batch[:-1], batch[-1], state)
            self.plot('loss', l, train=False)

        def apply_init(self, dummy_input, key):
            """To be defined later in :numref:`sec_lazy_init`"""
            raise NotImplementedError

    def configure_optimizers(self):
        raise NotImplementedError
```

```
class Module(d2l.nn_Module, d2l.HyperParameters):  #@save
    """The base class of models."""
    if tab.selected('mxnet', 'tensorflow'):
        def __init__(self, plot_train_per_epoch=2, plot_valid_per_epoch=1):
            super().__init__()
            self.save_hyperparameters()
            self.board = ProgressBoard()
        if tab.selected('tensorflow'):
            self.training = None

    if tab.selected('jax'):
        # No need for save_hyperparam when using Python dataclass
        plot_train_per_epoch: int = field(default=2, init=False)
        plot_valid_per_epoch: int = field(default=1, init=False)
        # Use default_factory to make sure new plots are generated on each run
        board: ProgressBoard = field(default_factory=lambda: ProgressBoard(),
                                     init=False)

    def loss(self, y_hat, y):
        raise NotImplementedError

    if tab.selected('mxnet', 'tensorflow'):
        def forward(self, X):
            assert hasattr(self, 'net'), 'Neural network is defined'
            return self.net(X)

    if tab.selected('tensorflow'):
        def call(self, X, *args, **kwargs):
            if kwargs and "training" in kwargs:
                self.training = kwargs['training']
            return self.forward(X, *args)

    if tab.selected('jax'):
        # JAX & Flax do not have a forward-method-like syntax. Flax uses setup
        # and built-in __call__ magic methods for forward pass. Adding here
        # for consistency
```

(continues on next page)

(continued from previous page)

```
        def forward(self, X, *args, **kwargs):
            assert hasattr(self, 'net'), 'Neural network is defined'
            return self.net(X, *args, **kwargs)

        def __call__(self, X, *args, **kwargs):
            return self.forward(X, *args, **kwargs)

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
        if tab.selected('mxnet', 'tensorflow'):
            self.board.draw(x, d2l.numpy(value), (
                'train_' if train else 'val_') + key, every_n=int(n))
        if tab.selected('jax'):
            self.board.draw(x, d2l.to(value, d2l.cpu()),
                            ('train_' if train else 'val_') + key,
                            every_n=int(n))

    if tab.selected('mxnet', 'tensorflow'):
        def training_step(self, batch):
            l = self.loss(self(*batch[:-1]), batch[-1])
            self.plot('loss', l, train=True)
            return l

        def validation_step(self, batch):
            l = self.loss(self(*batch[:-1]), batch[-1])
            self.plot('loss', l, train=False)

    if tab.selected('jax'):
        def training_step(self, params, batch, state):
            l, grads = jax.value_and_grad(self.loss)(params, batch[:-1],
                                                     batch[-1], state)
            self.plot("loss", l, train=True)
            return l, grads

        def validation_step(self, params, batch, state):
            l = self.loss(params, batch[:-1], batch[-1], state)
            self.plot('loss', l, train=False)

        def apply_init(self, dummy_input, key):
            """To be defined later in :numref:`sec_lazy_init`"""
            raise NotImplementedError

    def configure_optimizers(self):
        raise NotImplementedError
```

```
class Module(d2l.nn_Module, d2l.HyperParameters):  #@save
    """The base class of models."""
    if tab.selected('mxnet', 'tensorflow'):
        def __init__(self, plot_train_per_epoch=2, plot_valid_per_epoch=1):
            super().__init__()
            self.save_hyperparameters()
            self.board = ProgressBoard()
        if tab.selected('tensorflow'):
            self.training = None

    if tab.selected('jax'):
        # No need for save_hyperparam when using Python dataclass
        plot_train_per_epoch: int = field(default=2, init=False)
        plot_valid_per_epoch: int = field(default=1, init=False)
        # Use default_factory to make sure new plots are generated on each run
        board: ProgressBoard = field(default_factory=lambda: ProgressBoard(),
                                     init=False)

    def loss(self, y_hat, y):
        raise NotImplementedError

    if tab.selected('mxnet', 'tensorflow'):
        def forward(self, X):
            assert hasattr(self, 'net'), 'Neural network is defined'
            return self.net(X)

    if tab.selected('tensorflow'):
        def call(self, X, *args, **kwargs):
            if kwargs and "training" in kwargs:
                self.training = kwargs['training']
            return self.forward(X, *args)

    if tab.selected('jax'):
        # JAX & Flax do not have a forward-method-like syntax. Flax uses setup
        # and built-in __call__ magic methods for forward pass. Adding here
        # for consistency
        def forward(self, X, *args, **kwargs):
            assert hasattr(self, 'net'), 'Neural network is defined'
            return self.net(X, *args, **kwargs)

        def __call__(self, X, *args, **kwargs):
            return self.forward(X, *args, **kwargs)

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
```

(continues on next page)

(continued from previous page)

```
        if tab.selected('mxnet', 'tensorflow'):
            self.board.draw(x, d2l.numpy(value), (
                'train_' if train else 'val_') + key, every_n=int(n))
        if tab.selected('jax'):
            self.board.draw(x, d2l.to(value, d2l.cpu()),
                            ('train_' if train else 'val_') + key,
                            every_n=int(n))

    if tab.selected('mxnet', 'tensorflow'):
        def training_step(self, batch):
            l = self.loss(self(*batch[:-1]), batch[-1])
            self.plot('loss', l, train=True)
            return l

        def validation_step(self, batch):
            l = self.loss(self(*batch[:-1]), batch[-1])
            self.plot('loss', l, train=False)

    if tab.selected('jax'):
        def training_step(self, params, batch, state):
            l, grads = jax.value_and_grad(self.loss)(params, batch[:-1],
                                                     batch[-1], state)
            self.plot("loss", l, train=True)
            return l, grads

        def validation_step(self, params, batch, state):
            l = self.loss(params, batch[:-1], batch[-1], state)
            self.plot('loss', l, train=False)

        def apply_init(self, dummy_input, key):
            """To be defined later in :numref:`sec_lazy_init`"""
            raise NotImplementedError

    def configure_optimizers(self):
        raise NotImplementedError
```

Module が PyTorch のニューラルネットワークの基底クラスである nn.Module のサブクラスであることに気づくかもしれない。これはニューラルネットワークを扱うための便利な機能を提供する。たとえば、forward(self, X) のような forward メソッドを定義すると、インスタンス a に対して a(X) と書くだけでこのメソッドを呼び出せる。これは、組み込みの __call__ メソッドが forward を呼び出すためである。nn.Module についての詳細や例は Section 9.1 を参照されたい。

6.2.3 データ

DataModule クラスはデータのための基底クラスである。かなり頻繁に、__init__ メソッドはデータの準備に使われる。必要ならダウンロードや前処理も含まれる。train_dataloader は学習データセット用のデータローダーを返す。データローダーは、使われるたびにデータバッチを 1 つ返す（Python の）ジェネレータである。このバッチは Module の training_step メソッドに渡され、損失を計

算する。検証データセット用のローダーを返す val_dataloader も任意で用意できる。こちらも同様に動作するが、Module の validation_step メソッドに渡すデータバッチを返す。

```
class DataModule(d2l.HyperParameters):  #@save
    """The base class of data."""
    if tab.selected('mxnet', 'pytorch'):
        def __init__(self, root='../data', num_workers=4):
            self.save_hyperparameters()

    if tab.selected('tensorflow', 'jax'):
        def __init__(self, root='../data'):
            self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError

    def train_dataloader(self):
        return self.get_dataloader(train=True)

    def val_dataloader(self):
        return self.get_dataloader(train=False)
```

6.2.4 学習

Trainer クラスは、DataModule で指定されたデータを使って Module クラスの学習可能パラメータを学習する。中心となるメソッドは fit で、2 つの引数を受け取る。model は Module のインスタンス、data は DataModule のインスタンスである。その後、データセット全体を max_epochs 回繰り返してモデルを学習する。これまでと同様、このメソッドの実装は後の章に回す。

```
class Trainer(d2l.HyperParameters):  #@save
    """The base class for training models with data."""
    def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        self.save_hyperparameters()
        assert num_gpus == 0, 'No GPU support yet'

    def prepare_data(self, data):
        self.train_dataloader = data.train_dataloader()
        self.val_dataloader = data.val_dataloader()
        self.num_train_batches = len(self.train_dataloader)
        self.num_val_batches = (len(self.val_dataloader)
                                if self.val_dataloader is not None else 0)

    def prepare_model(self, model):
        model.trainer = self
        model.board.xlim = [0, self.max_epochs]
        self.model = model

    if tab.selected('pytorch', 'mxnet', 'tensorflow'):
        def fit(self, model, data):
            self.prepare_data(data)
            self.prepare_model(model)
```

(continues on next page)

(continued from previous page)

```
            self.optim = model.configure_optimizers()
            self.epoch = 0
            self.train_batch_idx = 0
            self.val_batch_idx = 0
            for self.epoch in range(self.max_epochs):
                self.fit_epoch()

    if tab.selected('jax'):
        def fit(self, model, data, key=None):
            self.prepare_data(data)
            self.prepare_model(model)
            self.optim = model.configure_optimizers()

            if key is None:
                root_key = d2l.get_key()
            else:
                root_key = key
            params_key, dropout_key = jax.random.split(root_key)
            key = {'params': params_key, 'dropout': dropout_key}

            dummy_input = next(iter(self.train_dataloader))[:-1]
            variables = model.apply_init(dummy_input, key=key)
            params = variables['params']

            if 'batch_stats' in variables.keys():
                # Here batch_stats will be used later (e.g., for batch norm)
                batch_stats = variables['batch_stats']
            else:
                batch_stats = {}

            # Flax uses optax under the hood for a single state obj TrainState.
            # More will be discussed later in the dropout and batch
            # normalization section
            class TrainState(train_state.TrainState):
                batch_stats: Any
                dropout_rng: jax.random.PRNGKeyArray

            self.state = TrainState.create(apply_fn=model.apply,
                                           params=params,
                                           batch_stats=batch_stats,
                                           dropout_rng=dropout_key,
                                           tx=model.configure_optimizers())
            self.epoch = 0
            self.train_batch_idx = 0
            self.val_batch_idx = 0
            for self.epoch in range(self.max_epochs):
                self.fit_epoch()

    def fit_epoch(self):
        raise NotImplementedError
```

6.2.5 要約

深層学習の今後の実装に向けたオブジェクト指向設計を強調するために、上のクラス群は、オブジェクトがどのようにデータを保存し、互いにやり取りするかを示しているにすぎない。この本の残りでは、`@add_to_class` などを通じて、これらのクラスの実装をさらに充実させていく。さらに、これらの完全実装済みクラスは D2L ライブラリ[55] に保存されており、深層学習の構造化モデリングを容易にする軽量ツールキットである。特に、プロジェクト間で多くの構成要素をほとんど変更せずに再利用できるようにする。たとえば、最適化手法だけ、モデルだけ、データセットだけを置き換えることができる。この程度のモジュール性は、本書全体を通して簡潔さと単純さの面で大きな効果をもたらす（そのためにこれを追加した）。そして、あなた自身のプロジェクトでも同じ効果を発揮するだろう。


55

6.2.6 演習

1. D2L ライブラリ[56] に保存されている上記クラスの完全実装を見つけよ。深層学習モデリングにもう少し慣れたら、実装を詳しく読むことを強く勧める。


56

2. `B` クラスの `save_hyperparameters` 文を削除せよ。それでも `self.a` と `self.b` を表示できるか？任意: `HyperParameters` クラスの完全実装まで読み進めたなら、なぜそうなるのか説明できるか？

6.3 合成回帰データ

機械学習とは、データから情報を抽出することにほかならない。では、合成データからいったい何を学べるのだろうか。人工的なデータ生成モデルに自分たちで組み込んだパターンそのものには本質的な関心がないとしても、そのようなデータセットは教育目的には非常に有用であり、学習アルゴリズムの性質を評価したり、実装が期待どおりに動作していることを確認したりするのに役立ちる。たとえば、正しいパラメータが *a priori* に既知であるデータを作成すれば、モデルがそれらを実際に復元できるかどうかを確認できる。

```
%matplotlib inline
from d2l import torch as d2l
import torch
import random
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx, gluon
import random
npx.set_np()
```

```
%matplotlib inline
from d2l import jax as d2l
import jax
from jax import numpy as jnp
import numpy as np
import random
import tensorflow as tf
import tensorflow_datasets as tfds
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf
import random
```

6.3.1 データセットの生成

この例では、簡潔さのために低次元の設定を扱いる。次のコード片は、標準正規分布から生成した 2 次元特徴をもつ 1000 個の例を作成する。得られる設計行列 $\mathbf{X}$ は $\mathbb{R}^{1000\times 2}$ に属する。各ラベルは、真の値の線形関数を適用し、さらに各例ごとに独立同分布に生成した加法ノイズ $\boldsymbol{\epsilon}$ を加えることで生成する。

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \boldsymbol{\epsilon}. \tag{6.3.1}$$

便宜上、$\boldsymbol{\epsilon}$ は平均 $\mu = 0$、標準偏差 $\sigma = 0.01$ の正規分布から生成されると仮定する。オブジェクト指向設計のために、コードを `d2l.DataModule` のサブクラスの `__init__` メソッドに追加する（Section 6.2.3 で導入）。追加のハイパーパラメータを設定できるようにしておくのがよい実践である。これには `save_hyperparameters()` を用いる。`batch_size` は後で決定する。

```
class SyntheticRegressionData(d2l.DataModule):  #@save
    """ 線形回帰のための合成データ。"""
    def __init__(self, w, b, noise=0.01, num_train=1000, num_val=1000,
                 batch_size=32):
        super().__init__()
        self.save_hyperparameters()
        n = num_train + num_val
        if tab.selected('pytorch') or tab.selected('mxnet'):
            self.X = d2l.randn(n, len(w))
            noise = d2l.randn(n, 1) * noise
        if tab.selected('tensorflow'):
            self.X = tf.random.normal((n, w.shape[0]))
            noise = tf.random.normal((n, 1)) * noise
        if tab.selected('jax'):
            key = jax.random.PRNGKey(0)
            key1, key2 = jax.random.split(key)
            self.X = jax.random.normal(key1, (n, w.shape[0]))
```

(continues on next page)

(continued from previous page)

```
            noise = jax.random.normal(key2, (n, 1)) * noise
        self.y = d2l.matmul(self.X, d2l.reshape(w, (-1, 1))) + b + noise
```

以下では、真のパラメータを $\mathbf{w} = [2, -3.4]^\top$、$b = 4.2$ に設定する。後で、推定したパラメータをこれらの 真の値と比較できる。

```
data = SyntheticRegressionData(w=d2l.tensor([2, -3.4]), b=4.2)
```

```
[07:18:05] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

features の各行は $\mathbb{R}^2$ のベクトルからなり、labels の各行はスカラーである。最初の要素を見てみよう。

```
print('features:', data.X[0],'\nlabel:', data.y[0])
```

```
features: tensor([ 0.1966, -0.6143])
label: tensor([6.6860])
```

6.3.2 データセットの読み出し

機械学習モデルの学習では、しばしばデータセットを何度も走査し、1 回に 1 つのミニバッチずつ取り出す。そのデータを使ってモデルを更新する。これがどのように機能するかを示すために、get_dataloader メソッドを実装し、add_to_class を介して SyntheticRegressionData クラスに登録する（Section 6.2.1 で導入）。これはバッチサイズ、特徴の行列、ラベルのベクトルを受け取り、サイズが batch_size のミニバッチを生成する。したがって、各ミニバッチは特徴とラベルのタプルからなる。訓練モードか検証モードかに注意する必要があることに留意されたい。前者ではデータをランダム順に読み出したい一方で、後者では、あらかじめ定めた順序でデータを読み出せることがデバッグ上重要になる場合がある。

```
@d2l.add_to_class(SyntheticRegressionData)
def get_dataloader(self, train):
    if train:
        indices = list(range(0, self.num_train))
        # The examples are read in random order
        random.shuffle(indices)
    else:
        indices = list(range(self.num_train, self.num_train+self.num_val))
    for i in range(0, len(indices), self.batch_size):
        if tab.selected('mxnet', 'pytorch', 'jax'):
            batch_indices = d2l.tensor(indices[i: i+self.batch_size])
            yield self.X[batch_indices], self.y[batch_indices]
        if tab.selected('tensorflow'):
```

(continues on next page)

(continued from previous page)

```
            j = tf.constant(indices[i : i+self.batch_size])
            yield tf.gather(self.X, j), tf.gather(self.y, j)
```

直感をつかむために、最初のミニバッチのデータを見てみよう。特徴の各ミニバッチは、そのサイズと入力特徴の次元数の両方を私たちに与える。同様に、ラベルのミニバッチも batch_size によって決まる対応する形状を持つ。

```
X, y = next(iter(data.train_dataloader()))
print('X shape:', X.shape, '\ny shape:', y.shape)
```

```
X shape: torch.Size([32, 2])
y shape: torch.Size([32, 1])
```

一見すると何でもないようであるが、iter(data.train_dataloader()) の呼び出しは、Python のオブジェクト指向設計の力を示している。data オブジェクトを作成した後で、SyntheticRegressionData クラスにメソッドを追加したことに注意されたい。それにもかかわらず、このオブジェクトはクラスへの機能追加を事後的に受けられる。

反復の間、データセット全体を使い切るまで異なるミニバッチが得られる（試されたい）。上で実装した反復処理は教育目的にはよいものの、実際の問題では困るような非効率さがある。たとえば、すべてのデータをメモリに読み込む必要があり、大量のランダムメモリアクセスを行わなければならない。深層学習フレームワークに実装されている組み込みイテレータははるかに効率的であり、ファイルに保存されたデータ、ストリーム経由で受け取るデータ、その場で生成または処理されるデータなどにも対応できる。次に、組み込みイテレータを使って同じメソッドを実装してみよう。

6.3.3 データローダの簡潔な実装

自前でイテレータを書く代わりに、フレームワークの既存 API を呼び出してデータを読み込むことができる。これまでと同様に、特徴 X とラベル y をもつデータセットが必要である。それに加えて、組み込みのデータローダで batch_size を設定し、例のシャッフルはフレームワークに任せることで効率化する。

```
@d2l.add_to_class(d2l.DataModule)  #@save
def get_tensorloader(self, tensors, train, indices=slice(0, None)):
    tensors = tuple(a[indices] for a in tensors)
    if tab.selected('mxnet'):
        dataset = gluon.data.ArrayDataset(*tensors)
        return gluon.data.DataLoader(dataset, self.batch_size,
                                     shuffle=train)
    if tab.selected('pytorch'):
        dataset = torch.utils.data.TensorDataset(*tensors)
        return torch.utils.data.DataLoader(dataset, self.batch_size,
```

(continues on next page)

(continued from previous page)

```
                                            shuffle=train)
    if tab.selected('jax'):
        # Use Tensorflow Datasets & Dataloader. JAX or Flax do not provide
        # any dataloading functionality
        shuffle_buffer = tensors[0].shape[0] if train else 1
        return tfds.as_numpy(
            tf.data.Dataset.from_tensor_slices(tensors).shuffle(
                buffer_size=shuffle_buffer).batch(self.batch_size))

    if tab.selected('tensorflow'):
        shuffle_buffer = tensors[0].shape[0] if train else 1
        return tf.data.Dataset.from_tensor_slices(tensors).shuffle(
            buffer_size=shuffle_buffer).batch(self.batch_size)
```

```
@d2l.add_to_class(SyntheticRegressionData)  #@save
def get_dataloader(self, train):
    i = slice(0, self.num_train) if train else slice(self.num_train, None)
    return self.get_tensorloader((self.X, self.y), train, i)
```

新しいデータローダは、より効率的でいくつかの追加機能を備えている点を除けば、前のものとまったく同じように動作する。

```
X, y = next(iter(data.train_dataloader()))
print('X shape:', X.shape, '\ny shape:', y.shape)
```

```
X shape: torch.Size([32, 2])
y shape: torch.Size([32, 1])
```

たとえば、フレームワーク API が提供するデータローダは組み込みの `__len__` メソッドをサポートしているので、その長さ、すなわちバッチ数を問い合わせることができる。

```
len(data.train_dataloader())
```

```
32
```

6.3.4 まとめ

データローダは、データの読み込みと操作の過程を抽象化する便利な方法である。これにより、同じ機械学習アルゴリズムが、変更を加えることなく多様な種類や供給源のデータを処理できるようになる。データローダのよい点の 1 つは、組み合わせ可能であることである。たとえば、画像を読み込んだあとに、それらを切り抜いたり他の方法で変更したりする後処理フィルタを適用することができる。このように、データローダはデータ処理パイプライン全体を記述するために使える。

モデル自体について言えば、2 次元線形モデルは私たちが出会う中で最も単純なものの 1 つである。これにより、データ不足や不定な連立方程式を心配することなく、回帰モデルの精度を試すことができる。次の節でこれを有効に活用する。

6.3.5 演習

1. 例の数がバッチサイズで割り切れない場合はどうなるか。フレームワークの API を使って別の引数を指定することで、この挙動をどのように変更するか？
2. パラメータベクトル `w` のサイズも例の数 `num_examples` も大きい、巨大なデータセットを生成したいとする。
   1. すべてのデータをメモリに保持できない場合、何が起こりますか。
   2. データがディスク上にある場合、どのようにシャッフルするか。あまり多くのランダム読み書きを必要としない、効率的なアルゴリズムを設計するのが課題である。ヒント: 擬似ランダム置換生成器[57] を使うと、置換表を明示的に保存せずに再シャッフルを設計できる (Naor and Reingold, 1999)。
3. イテレータが呼ばれるたびに、その場で新しいデータを生成するデータ生成器を実装せよ。
4. 呼ばれるたびに 同じデータを生成する乱数データ生成器をどのように設計するか？

[57]

6.4 線形回帰のスクラッチ実装

ここで、線形回帰の完全に動作する実装に取り組む準備が整った。

この節では、(i) モデル、(ii) 損失関数、(iii) ミニバッチ確率勾配降下法オプティマイザ、(iv) これらすべてをつなぎ合わせる学習関数を含め、方法全体をスクラッチから実装する。

最後に、Section 6.3 の合成データ生成器を実行し、その結果得られるデータセットに対してモデルを適用する。

現代の深層学習フレームワークはこの作業のほとんどを自動化できるが、スクラッチから実装することこそが、自分が何をしているのかを本当に理解していることを確かめる唯一の方法である。

さらに、モデルをカスタマイズしたり、自前の層や損失関数を定義したりするときには、内部で何が起きているかを理解していることが大いに役立ちる。

この節では、テンソルと自動微分だけに頼りる。

後の節では、以下の構造を保ちながら、深層学習フレームワークの便利な機能を活用した、より簡潔な実装を紹介する。

```
%matplotlib inline
from d2l import torch as d2l
import torch
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, np, npx
npx.set_np()
```

```
%matplotlib inline
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
import optax
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf
```

6.4.1 モデルの定義

ミニバッチ SGD でモデルのパラメータを最適化し始める前に、まずパラメータが必要である。

以下では、平均 0、標準偏差 0.01 の正規分布から乱数を引くことで重みを初期化する。

魔法の数 0.01 は実践上しばしばうまくいくが、引数 `sigma` を通して別の値を指定することもできる。

さらに、バイアスは 0 に設定する。

オブジェクト指向設計においては、コードを `d2l.Module` のサブクラスの `__init__` メソッドに追加する（Section 6.2.2 で導入）。

```
class LinearRegressionScratch(d2l.Module):  #@save
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.w = d2l.normal(0, sigma, (num_inputs, 1))
            self.b = d2l.zeros(1)
            self.w.attach_grad()
```

(continues on next page)

(continued from previous page)

```
            self.b.attach_grad()
        if tab.selected('pytorch'):
            self.w = d2l.normal(0, sigma, (num_inputs, 1), requires_grad=True)
            self.b = d2l.zeros(1, requires_grad=True)
        if tab.selected('tensorflow'):
            w = tf.random.normal((num_inputs, 1), mean=0, stddev=0.01)
            b = tf.zeros(1)
            self.w = tf.Variable(w, trainable=True)
            self.b = tf.Variable(b, trainable=True)
```

```
class LinearRegressionScratch(d2l.Module):  #@save
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.w = d2l.normal(0, sigma, (num_inputs, 1))
            self.b = d2l.zeros(1)
            self.w.attach_grad()
            self.b.attach_grad()
        if tab.selected('pytorch'):
            self.w = d2l.normal(0, sigma, (num_inputs, 1), requires_grad=True)
            self.b = d2l.zeros(1, requires_grad=True)
        if tab.selected('tensorflow'):
            w = tf.random.normal((num_inputs, 1), mean=0, stddev=0.01)
            b = tf.zeros(1)
            self.w = tf.Variable(w, trainable=True)
            self.b = tf.Variable(b, trainable=True)
```

```
class LinearRegressionScratch(d2l.Module):  #@save
    """The linear regression model implemented from scratch."""
    num_inputs: int
    lr: float
    sigma: float = 0.01

    def setup(self):
        self.w = self.param('w', nn.initializers.normal(self.sigma),
                            (self.num_inputs, 1))
        self.b = self.param('b', nn.initializers.zeros, (1))
```

```
class LinearRegressionScratch(d2l.Module):  #@save
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.w = d2l.normal(0, sigma, (num_inputs, 1))
            self.b = d2l.zeros(1)
            self.w.attach_grad()
            self.b.attach_grad()
        if tab.selected('pytorch'):
            self.w = d2l.normal(0, sigma, (num_inputs, 1), requires_grad=True)
```

(continues on next page)

(continued from previous page)

```
            self.b = d2l.zeros(1, requires_grad=True)
        if tab.selected('tensorflow'):
            w = tf.random.normal((num_inputs, 1), mean=0, stddev=0.01)
            b = tf.zeros(1)
            self.w = tf.Variable(w, trainable=True)
            self.b = tf.Variable(b, trainable=True)
```

次に、入力とパラメータを出力に結びつけるモデルを定義する必要がある。
(6.1.4) と同じ記法を用いると、線形モデルでは入力特徴量 $\mathbf{X}$ とモデルの重み $\mathbf{w}$ の行列–ベクトル積を取り、各例にオフセット b を加えるだけです。
積 $\mathbf{Xw}$ はベクトルであり、b はスカラーである。
ブロードキャスト機構（Section 5.1.4 を参照）により、ベクトルとスカラーを加えると、スカラーはベクトルの各成分に加えられる。
以下の `forward` メソッドは、`add_to_class`（Section 6.2.1 で導入）を通じて `LinearRegressionScratch` クラスに登録される。

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def forward(self, X):
    return d2l.matmul(X, self.w) + self.b
```

6.4.2 損失関数の定義

モデルの更新には損失関数の勾配を取る必要があるので、まず損失関数を定義しておくべきである。
ここでは (6.1.5) の二乗損失関数を使う。
実装では、真の値 `y` を予測値 `y_hat` の形状に変換する必要がある。
以下のメソッドが返す結果も `y_hat` と同じ形状になる。
また、ミニバッチ内の全例にわたる平均損失値も返す。

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return d2l.reduce_mean(l)
```


```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return d2l.reduce_mean(l)
```

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, params, X, y, state):
    y_hat = state.apply_fn({'params': params}, *X)  # X unpacked from a tuple
    l = (y_hat - d2l.reshape(y, y_hat.shape)) ** 2 / 2
    return d2l.reduce_mean(l)
```

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return d2l.reduce_mean(l)
```

6.4.3 最適化アルゴリズムの定義

Section 6.1 で議論したように、線形回帰には閉形式解がある。
しかし、ここでの目的は、より一般的なニューラルネットワークをどのように学習させるかを示すことである。そのためには、ミニバッチ SGD の使い方を学ぶ必要がある。
そこで、この機会に SGD の最初の動作例を紹介する。
各ステップで、データセットからランダムに取り出したミニバッチを用いて、パラメータに関する損失の勾配を推定する。
次に、損失を減らす方向にパラメータを更新する。

以下のコードは、パラメータ集合と学習率 `lr` が与えられたときに更新を適用する。
損失はミニバッチ上の平均として計算されるので、バッチサイズに応じて学習率を調整する必要はない。
後の章では、分散大規模学習で現れるような非常に大きなミニバッチに対して、学習率をどのように調整すべきかを調べる。
今のところは、この依存関係は無視して構わない。

`d2l.HyperParameters`（Section 6.2.1 で導入）のサブクラスとして `SGD` クラスを定義し、組み込みの SGD オプティマイザと同様の API を持たせる。
パラメータは `step` メソッドで更新する。
`zero_grad` メソッドはすべての勾配を 0 に設定し、逆伝播の前に実行しなければならない。

```
class SGD(d2l.HyperParameters):  #@save
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()
```

(continues on next page)

(continued from previous page)

```
    if tab.selected('mxnet'):
        def step(self, _):
            for param in self.params:
                param -= self.lr * param.grad

    if tab.selected('pytorch'):
        def step(self):
            for param in self.params:
                param -= self.lr * param.grad

        def zero_grad(self):
            for param in self.params:
                if param.grad is not None:
                    param.grad.zero_()
```

```
class SGD(d2l.HyperParameters):  #@save
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()

    if tab.selected('mxnet'):
        def step(self, _):
            for param in self.params:
                param -= self.lr * param.grad

    if tab.selected('pytorch'):
        def step(self):
            for param in self.params:
                param -= self.lr * param.grad

        def zero_grad(self):
            for param in self.params:
                if param.grad is not None:
                    param.grad.zero_()
```

```
class SGD(d2l.HyperParameters):  #@save
    """Minibatch stochastic gradient descent."""
    # The key transformation of Optax is the GradientTransformation
    # defined by two methods, the init and the update.
    # The init initializes the state and the update transforms the gradients.
    # https://github.com/deepmind/optax/blob/master/optax/_src/transform.py
    def __init__(self, lr):
        self.save_hyperparameters()

    def init(self, params):
        # Delete unused params
        del params
        return optax.EmptyState

    def update(self, updates, state, params=None):
        del params
        # When state.apply_gradients method is called to update flax's
```

(continues on next page)

(continued from previous page)

```
        # train_state object, it internally calls optax.apply_updates method
        # adding the params to the update equation defined below.
        updates = jax.tree_util.tree_map(lambda g: -self.lr * g, updates)
        return updates, state

    def __call__():
        return optax.GradientTransformation(self.init, self.update)
```

```
class SGD(d2l.HyperParameters):  #@save
    """Minibatch stochastic gradient descent."""
    def __init__(self, lr):
        self.save_hyperparameters()

    def apply_gradients(self, grads_and_vars):
        for grad, param in grads_and_vars:
            param.assign_sub(self.lr * grad)
```

次に、SGD クラスのインスタンスを返す configure_optimizers メソッドを定義する。

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def configure_optimizers(self):
    if tab.selected('mxnet') or tab.selected('pytorch'):
        return SGD([self.w, self.b], self.lr)
    if tab.selected('tensorflow', 'jax'):
        return SGD(self.lr)
```

6.4.4 学習

これで、すべての部品（パラメータ、損失関数、モデル、オプティマイザ）が揃ったので、主要な学習ループを実装する準備が整った。

このコードを完全に理解することは非常に重要である。というのも、この本で扱う他のすべての深層学習モデルでも、同様の学習ループを使うからである。

各 *epoch* では、訓練データセット全体を一巡し、すべての例を 1 回ずつ通過する（例の数がバッチサイズで割り切れると仮定する）。

各 *iteration* では、訓練例のミニバッチを取り出し、モデルの training_step メソッドを通して損失を計算する。

次に、各パラメータに関する勾配を計算する。

最後に、最適化アルゴリズムを呼び出してモデルパラメータを更新する。

要するに、以下のループを実行する。

- パラメータ $(\mathbf{w}, b)$ を初期化する
- 完了するまで繰り返す
  - 勾配 $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$ を計算する

-- パラメータ $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$ を更新する

Section 6.3 で生成した合成回帰データセットには検証データセットが含まれていないことを思い出そう。
しかし一般には、モデル品質を測定するために検証データセットが欲しくなる。
ここでは、各 epoch ごとに検証データローダーを 1 回通してモデル性能を測定する。
オブジェクト指向設計に従い、`prepare_batch` と `fit_epoch` メソッドは `d2l.Trainer` クラスに登録される（Section 6.2.4 で導入）。

```
@d2l.add_to_class(d2l.Trainer)  #@save
def prepare_batch(self, batch):
    return batch
```

```
@d2l.add_to_class(d2l.Trainer)  #@save
def fit_epoch(self):
    self.model.train()
    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0:  # To be discussed later
                self.clip_gradients(self.gradient_clip_val, self.model)
            self.optim.step()
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
    self.model.eval()
    for batch in self.val_dataloader:
        with torch.no_grad():
            self.model.validation_step(self.prepare_batch(batch))
        self.val_batch_idx += 1
```

```
@d2l.add_to_class(d2l.Trainer)  #@save
def fit_epoch(self):
    for batch in self.train_dataloader:
        with autograd.record():
            loss = self.model.training_step(self.prepare_batch(batch))
        loss.backward()
        if self.gradient_clip_val > 0:
            self.clip_gradients(self.gradient_clip_val, self.model)
        self.optim.step(1)
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
```

(continues on next page)

(continued from previous page)

```
    for batch in self.val_dataloader:
        self.model.validation_step(self.prepare_batch(batch))
        self.val_batch_idx += 1
```

```
@d2l.add_to_class(d2l.Trainer)  #@save
def fit_epoch(self):
    self.model.training = True
    if self.state.batch_stats:
        # Mutable states will be used later (e.g., for batch norm)
        for batch in self.train_dataloader:
            (_, mutated_vars), grads = self.model.training_step(self.state.
↪params,
                                                                self.prepare_
↪batch(batch),
                                                                self.state)
            self.state = self.state.apply_gradients(grads=grads)
            # Can be ignored for models without Dropout Layers
            self.state = self.state.replace(
                dropout_rng=jax.random.split(self.state.dropout_rng)[0])
            self.state = self.state.replace(batch_stats=mutated_vars['batch_
↪stats'])
            self.train_batch_idx += 1
    else:
        for batch in self.train_dataloader:
            _, grads = self.model.training_step(self.state.params,
                                                self.prepare_batch(batch),
                                                self.state)
            self.state = self.state.apply_gradients(grads=grads)
            # Can be ignored for models without Dropout Layers
            self.state = self.state.replace(
                dropout_rng=jax.random.split(self.state.dropout_rng)[0])
            self.train_batch_idx += 1

    if self.val_dataloader is None:
        return
    self.model.training = False
    for batch in self.val_dataloader:
        self.model.validation_step(self.state.params,
                                   self.prepare_batch(batch),
                                   self.state)
        self.val_batch_idx += 1
```

```
@d2l.add_to_class(d2l.Trainer)  #@save
def fit_epoch(self):
    self.model.training = True
    for batch in self.train_dataloader:
        with tf.GradientTape() as tape:
            loss = self.model.training_step(self.prepare_batch(batch))
        grads = tape.gradient(loss, self.model.trainable_variables)
        if self.gradient_clip_val > 0:
            grads = self.clip_gradients(self.gradient_clip_val, grads)
        self.optim.apply_gradients(zip(grads, self.model.trainable_variables))
        self.train_batch_idx += 1
```

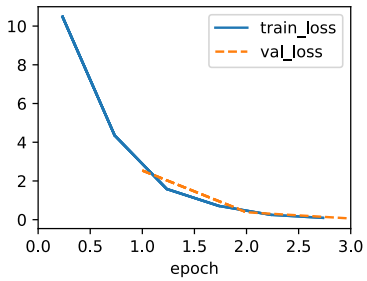
(continues on next page)

(continued from previous page)

```
    if self.val_dataloader is None:
        return
    self.model.training = False
    for batch in self.val_dataloader:
        self.model.validation_step(self.prepare_batch(batch))
        self.val_batch_idx += 1
```

モデルを学習する準備はほぼ整ったが、その前に学習データが必要である。
ここでは SyntheticRegressionData クラスを使い、真のパラメータを与える。
そして、学習率 lr=0.03 でモデルを学習し、max_epochs=3 に設定する。
一般に、epoch 数と学習率の両方はハイパーパラメータであることに注意されたい。
ハイパーパラメータの設定は一般に難しく、通常はデータセットを 3 つに分割したくなる。すなわち、1 つは学習用、2 つ目はハイパーパラメータ選択用の検証用、そして 3 つ目は最終評価用である。
ここではこれらの詳細は省きるが、後ほど改めて詳しく取り上げる。

```
model = LinearRegressionScratch(2, lr=0.03)
data = d2l.SyntheticRegressionData(w=d2l.tensor([2, -3.4]), b=4.2)
trainer = d2l.Trainer(max_epochs=3)
trainer.fit(model, data)
```



データセットを自分たちで合成したので、真のパラメータが何であるかを正確に知っている。
したがって、学習ループを通じて真のパラメータと学習したパラメータを比較することで、学習の成功を評価できる。
実際、それらは非常に近い値になる。

```
with torch.no_grad():
    print(f'error in estimating w: {data.w - d2l.reshape(model.w, data.w.
↪shape)}')
    print(f'error in estimating b: {data.b - model.b}')
```

```
error in estimating w: tensor([ 0.0709, -0.2553])
error in estimating b: tensor([0.2337])
```

```
print(f'error in estimating w: {data.w - d2l.reshape(model.w, data.w.shape)}')
print(f'error in estimating b: {data.b - model.b}')
```

```
error in estimating w: [ 0.1114291  -0.12024117]
error in estimating b: [0.1888752]
```

```
params = trainer.state.params
print(f"error in estimating w: {data.w - d2l.reshape(params['w'], data.w.
↪shape)}")
print(f"error in estimating b: {data.b - params['b']}")
```

```
error in estimating w: [ 0.07124913 -0.19058824]
error in estimating b: [0.23817086]
```

```
print(f'error in estimating w: {data.w - d2l.reshape(model.w, data.w.shape)}')
print(f'error in estimating b: {data.b - model.b}')
```

```
error in estimating w: [ 0.12973297 -0.16401815]
error in estimating b: [0.25387883]
```

真のパラメータを正確に復元できることを当然だと思うべきではない。
一般に、深いモデルではパラメータの一意な解は存在せず、線形モデルであっても、ある特徴量が他の特徴量に対して線形従属でない場合にのみ、パラメータを正確に復元できる。
しかし機械学習では、真の潜在パラメータを復元することよりも、高精度な予測をもたらすパラメータのほうが重要であることが多いです (Vapnik, 1992)。
幸いなことに、難しい最適化問題であっても、確率勾配降下法はしばしば驚くほど良い解を見つけられる。これは、深層ネットワークでは高精度な予測をもたらすパラメータの構成が多数存在することが、一因である。

6.4.5 まとめ

この節では、完全に機能するニューラルネットワークモデルと学習ループを実装することで、深層学習システムの設計に向けて大きな一歩を踏み出した。

この過程で、データローダー、モデル、損失関数、最適化手続き、可視化・監視ツールを構築した。

これは、モデル学習に関係するすべての要素を含む Python オブジェクトを組み立てることで実現した。

これはまだプロフェッショナル品質の実装ではないが、完全に機能し、このようなコードは小さな問題を素早く解くのにすでに役立ちる。

今後の節では、これを より簡潔に（定型コードを避けて）かつ より効率的に（GPU を最大限に活用して）行う方法を見ていく。

6.4.6 演習

1. 重みを 0 で初期化したらどうなるだろうか。アルゴリズムはそれでも動作するだろうか。では、パラメータを 0.01 ではなく分散 1000 で初期化したらどうだろうか。

2. 
[58] あなたが Georg Simon Ohm[58] で、電圧と電流を関連づける抵抗のモデルを考えようとしているとする。自動微分を使ってモデルのパラメータを学習できるか。

3. 
[59] プランクの法則[59] を使って、スペクトルエネルギー密度から物体の温度を求められるだろうか。参考までに、黒体から放射される放射のスペクトル密度 B は $B(\lambda, T) = \frac{2hc^2}{\lambda^5} \cdot \left(\exp \frac{hc}{\lambda kT} - 1\right)^{-1}$ である。ここで、λ は波長、T は温度、c は光速、h はプランク定数、k はボルツマン定数である。異なる波長 λ に対するエネルギーを測定したとして、スペクトル密度曲線をプランクの法則にフィットさせる必要がある。

4. 損失の 2 階微分を計算したい場合、どのような問題に遭遇する可能性があるか。それらをどう解決するか。

5. `loss` 関数で `reshape` メソッドが必要なのはなぜですか。

6. 異なる学習率を使って実験し、損失関数の値がどれくらい速く下がるかを調べよ。学習エポック数を増やすことで誤差を減らせますか。

7. 例の数がバッチサイズで割り切れない場合、epoch の終わりに `data_iter` はどうなるか。

8. 絶対値損失 `(y_hat - d2l.reshape(y, y_hat.shape)).abs().sum()` のような別の損失関数を実装してみよ。

   1. 通常のデータで何が起こるか確認せよ。

   2. $mathbf{y}$ のいくつかの要素、たとえば $y_5 = 10000$ を意図的に摂動させた場合、挙動に違いがあるか確認せよ。

3. 二乗損失と絶対値損失の長所を組み合わせる安価な解決策を考えられますか。ヒント: 非常に大きな勾配値をどう避けますか。

9. なぜデータセットを再シャッフルする必要があるのだろうか。悪意を持って構成されたデータセットが、そうしないと最適化アルゴリズムを破綻させるような例を設計できるか。

6.5 線形回帰の簡潔な実装

深層学習は、この 10 年で一種のカンブリア爆発を経験してきた。技術、応用、アルゴリズムの数は、過去数十年の進歩をはるかに上回っている。これは複数の要因が幸運にも組み合わさった結果であり、その一つが、いくつかのオープンソース深層学習フレームワークが提供する強力で無料のツールである。Theano (Bergstra *et al.*, 2010), DistBelief (Dean *et al.*, 2012), および Caffe (Jia *et al.*, 2014) は、広く採用されたそのようなモデルの第一世代を代表していると言えるだろう。Lisp 風のプログラミング体験を提供した SN2 (Simulateur Neuristique) (Bottou and Le Cun, 1988) のような初期の（先駆的な）研究とは対照的に、現代のフレームワークは自動微分と Python の利便性を提供する。これらのフレームワークにより、勾配ベース学習アルゴリズムの実装における反復的な作業を自動化し、モジュール化できる。

Section 6.4 では、(i) データ保存と線形代数のためのテンソル、および (ii) 勾配計算のための自動微分だけに依拠した。実際には、データイテレータ、損失関数、最適化器、ニューラルネットワーク層は非常に一般的であるため、現代のライブラリはこれらの構成要素も私たちの代わりに実装してくれる。この節では、深層学習フレームワークの高レベル API を使って Section 6.4 の線形回帰モデルを簡潔に実装する方法を示す。

```
from d2l import torch as d2l
import numpy as np
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
import optax
```

```
from d2l import tensorflow as d2l
import numpy as np
import tensorflow as tf
```

6.5.1 モデルの定義

Section 6.4 で線形回帰をスクラッチから実装したときには、モデルパラメータを明示的に定義し、基本的な線形代数演算を使って出力を生成する計算をコード化した。これは知っておくべきことである。しかし、モデルがより複雑になり、しかもそれをほぼ毎日行わなければならなくなると、助けがあることのありがたさが分かるだろう。状況は、ブログをスクラッチから自作するのに似ている。一度や二度ならやりがいがあり、学びもあるが、車輪の再発明に 1 か月費やすようでは、優秀な Web 開発者とは言えない。

標準的な演算については、フレームワークにあらかじめ定義された層を使うことができ、実装を気にするよりも、モデルを構成する層そのものに集中できる。Fig. 6.1.2 で説明した単層ネットワークのアーキテクチャを思い出そう。この層は 全結合と呼ばれる。なぜなら、その入力の各要素が行列—ベクトル積によって各出力に接続されているからである。

PyTorch では、全結合層は `Linear` クラスと `LazyLinear` クラス（バージョン 1.8.0 以降で利用可能）で定義される。後者は、ユーザーが単に出力次元だけを指定できるようにするが、前者ではそれに加えて、この層に何個の入力が入るかも指定する必要がある。入力形状の指定は不便であり、（畳み込み層のように）非自明な計算を要することがある。そのため、簡潔さのために、可能な限りこのような「遅延」層を使う。

```
class LinearRegression(d2l.Module):  #@save
    """The linear regression model implemented with high-level APIs."""
    def __init__(self, lr):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Dense(1)
            self.net.initialize(init.Normal(sigma=0.01))
        if tab.selected('tensorflow'):
            initializer = tf.initializers.RandomNormal(stddev=0.01)
            self.net = tf.keras.layers.Dense(1, kernel_initializer=initializer)
        if tab.selected('pytorch'):
            self.net = nn.LazyLinear(1)
            self.net.weight.data.normal_(0, 0.01)
            self.net.bias.data.fill_(0)
```

```
class LinearRegression(d2l.Module):  #@save
    """The linear regression model implemented with high-level APIs."""
    def __init__(self, lr):
        super().__init__()
        self.save_hyperparameters()
```

(continues on next page)

(continued from previous page)

```
        if tab.selected('mxnet'):
            self.net = nn.Dense(1)
            self.net.initialize(init.Normal(sigma=0.01))
        if tab.selected('tensorflow'):
            initializer = tf.initializers.RandomNormal(stddev=0.01)
            self.net = tf.keras.layers.Dense(1, kernel_initializer=initializer)
        if tab.selected('pytorch'):
            self.net = nn.LazyLinear(1)
            self.net.weight.data.normal_(0, 0.01)
            self.net.bias.data.fill_(0)
```

```
class LinearRegression(d2l.Module):  #@save
    """The linear regression model implemented with high-level APIs."""
    lr: float

    def setup(self):
        self.net = nn.Dense(1, kernel_init=nn.initializers.normal(0.01))
```

```
class LinearRegression(d2l.Module):  #@save
    """The linear regression model implemented with high-level APIs."""
    def __init__(self, lr):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Dense(1)
            self.net.initialize(init.Normal(sigma=0.01))
        if tab.selected('tensorflow'):
            initializer = tf.initializers.RandomNormal(stddev=0.01)
            self.net = tf.keras.layers.Dense(1, kernel_initializer=initializer)
        if tab.selected('pytorch'):
            self.net = nn.LazyLinear(1)
            self.net.weight.data.normal_(0, 0.01)
            self.net.bias.data.fill_(0)
```

forward メソッドでは、あらかじめ定義された層の組み込み __call__ メソッドを呼び出して出力を計算するだけである。

```
@d2l.add_to_class(LinearRegression)  #@save
def forward(self, X):
    return self.net(X)
```

6.5.2 損失関数の定義

MSELoss クラスは平均二乗誤差を計算する（(6.1.5) の 1/2 因子は含まない）。デフォルトでは、MSELoss はサンプル全体の平均損失を返す。自前で実装するよりも高速で（しかも使いやすいです）。

```
@d2l.add_to_class(LinearRegression)  #@save
def loss(self, y_hat, y):
```

(continues on next page)

(continued from previous page)

```
    if tab.selected('mxnet'):
        fn = gluon.loss.L2Loss()
        return fn(y_hat, y).mean()
    if tab.selected('pytorch'):
        fn = nn.MSELoss()
        return fn(y_hat, y)
    if tab.selected('tensorflow'):
        fn = tf.keras.losses.MeanSquaredError()
        return fn(y, y_hat)
```

```
@d2l.add_to_class(LinearRegression)  #@save
def loss(self, y_hat, y):
    if tab.selected('mxnet'):
        fn = gluon.loss.L2Loss()
        return fn(y_hat, y).mean()
    if tab.selected('pytorch'):
        fn = nn.MSELoss()
        return fn(y_hat, y)
    if tab.selected('tensorflow'):
        fn = tf.keras.losses.MeanSquaredError()
        return fn(y, y_hat)
```

```
@d2l.add_to_class(LinearRegression)  #@save
def loss(self, params, X, y, state):
    y_hat = state.apply_fn({'params': params}, *X)
    return d2l.reduce_mean(optax.l2_loss(y_hat, y))
```

```
@d2l.add_to_class(LinearRegression)  #@save
def loss(self, y_hat, y):
    if tab.selected('mxnet'):
        fn = gluon.loss.L2Loss()
        return fn(y_hat, y).mean()
    if tab.selected('pytorch'):
        fn = nn.MSELoss()
        return fn(y_hat, y)
    if tab.selected('tensorflow'):
        fn = tf.keras.losses.MeanSquaredError()
        return fn(y, y_hat)
```

6.5.3 最適化アルゴリズムの定義

ミニバッチ SGD はニューラルネットワークを最適化するための標準的な手法であり、そのため PyTorch は `optim` モジュールでこのアルゴリズムのいくつかの変種をサポートしている。`SGD` インスタンスを生成するときには、最適化対象のパラメータ（モデルの `self.parameters()` から取得可能）と、最適化アルゴリズムに必要な学習率（`self.lr`）を指定する。

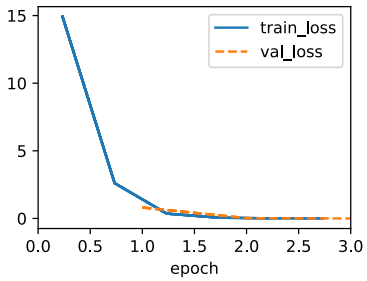
```
@d2l.add_to_class(LinearRegression)  #@save
def configure_optimizers(self):
    if tab.selected('mxnet'):
        return gluon.Trainer(self.collect_params(),
                             'sgd', {'learning_rate': self.lr})
    if tab.selected('pytorch'):
        return torch.optim.SGD(self.parameters(), self.lr)
    if tab.selected('tensorflow'):
        return tf.keras.optimizers.SGD(self.lr)
    if tab.selected('jax'):
        return optax.sgd(self.lr)
```

6.5.4 学習

深層学習フレームワークの高レベル API を通してモデルを表現すると、必要なコード行数が少なくなることに気づいたかもしれない。パラメータを個別に割り当てたり、損失関数を定義したり、ミニバッチ SGD を実装したりする必要はなかった。より複雑なモデルを扱い始めると、高レベル API の利点はさらに大きくなる。

基本要素がすべて揃ったので、学習ループ自体はスクラッチ実装したものと同じである。したがって、`fit` メソッド（Section 6.2.4 で導入）を呼び出すだけで、Section 6.4 の `fit_epoch` メソッドの実装に依存して、モデルを学習できる。

```
model = LinearRegression(lr=0.03)
data = d2l.SyntheticRegressionData(w=d2l.tensor([2, -3.4]), b=4.2)
trainer = d2l.Trainer(max_epochs=3)
trainer.fit(model, data)
```



以下では、有限データで学習して得られたモデルパラメータと実際のパラメータを比較する。パラメータにアクセスするには、必要な層の重みとバイアスにアクセスする。スクラッチ実装の場合と同様に、推定されたパラメータが真の値に近いことに注意されたい。

```
@d2l.add_to_class(LinearRegression)  #@save
def get_w_b(self):
```

(continues on next page)

(continued from previous page)

```
    if tab.selected('mxnet'):
        return (self.net.weight.data(), self.net.bias.data())
    if tab.selected('pytorch'):
        return (self.net.weight.data, self.net.bias.data)
    if tab.selected('tensorflow'):
        return (self.get_weights()[0], self.get_weights()[1])

w, b = model.get_w_b()
```

```
@d2l.add_to_class(LinearRegression)  #@save
def get_w_b(self):
    if tab.selected('mxnet'):
        return (self.net.weight.data(), self.net.bias.data())
    if tab.selected('pytorch'):
        return (self.net.weight.data, self.net.bias.data)
    if tab.selected('tensorflow'):
        return (self.get_weights()[0], self.get_weights()[1])

w, b = model.get_w_b()
```

```
@d2l.add_to_class(LinearRegression)  #@save
def get_w_b(self, state):
    net = state.params['net']
    return net['kernel'], net['bias']

w, b = model.get_w_b(trainer.state)
```

```
@d2l.add_to_class(LinearRegression)  #@save
def get_w_b(self):
    if tab.selected('mxnet'):
        return (self.net.weight.data(), self.net.bias.data())
    if tab.selected('pytorch'):
        return (self.net.weight.data, self.net.bias.data)
    if tab.selected('tensorflow'):
        return (self.get_weights()[0], self.get_weights()[1])

w, b = model.get_w_b()
```

```
print(f'error in estimating w: {data.w - d2l.reshape(w, data.w.shape)}')
print(f'error in estimating b: {data.b - b}')
```

```
error in estimating w: tensor([ 0.0080, -0.0134])
error in estimating b: tensor([0.0148])
```

6.5.5 まとめ

この節では、本書で初めて、MXNet (Chen *et al.*, 2015)、JAX (Frostig *et al.*, 2018)、PyTorch (Paszke *et al.*, 2019)、および Tensorflow (Abadi *et al.*, 2016) のような現代の

深層学習フレームワークがもたらす利便性を活用した深層ネットワークの実装を行った。データの読み込み、層の定義、損失関数、最適化器、学習ループにはフレームワークのデフォルトを使った。フレームワークが必要な機能をすべて提供しているなら、通常はそれらを使うのがよいだろう。これらの構成要素のライブラリ実装は性能のために大きく最適化されており、信頼性のために適切にテストされている傾向があるからである。同時に、これらのモジュールは直接実装できることも忘れないようにしたい。これは特に、モデル開発の最前線で生きたいと願う意欲的な研究者にとって重要である。そこでは、現在のどのライブラリにも存在しえない新しい構成要素を発明することになるからである。

PyTorch では、`data` モジュールがデータ処理のためのツールを提供し、`nn` モジュールが多数のニューラルネットワーク層と一般的な損失関数を定義する。末尾が _ のメソッドで値を置き換えることで、パラメータを初期化できる。ネットワークの入力次元を指定する必要があることに注意されたい。今のところは単純であるが、多数の層を持つ複雑なネットワークを設計したいときには、大きな波及効果を持つ可能性がある。これらのネットワークをどのようにパラメータ化するかを慎重に考える必要があり、それによって移植性を確保できる。

6.5.6 演習

1. ミニバッチ上の損失の総和を使う代わりに、ミニバッチ上の損失の平均を使うようにした場合、学習率はどのように変更する必要があるか？

2. フレームワークのドキュメントを確認して、どの損失関数が提供されているかを見よ。特に、二乗損失を Huber のロバスト損失関数に置き換えよ。すなわち、次の損失関数を使う。

$$l(y, y') = \begin{cases} |y - y'| - \frac{\sigma}{2} & \text{if } |y - y'| > \sigma \\ \frac{1}{2\sigma}(y - y')^2 & \text{otherwise} \end{cases} \tag{6.5.1}$$

3. モデルの重みの勾配にはどのようにアクセスするか？

4. 学習率とエポック数を変えると、解にどのような影響があるか？改善し続けますか？

5. 生成するデータ量を変えると、解はどのように変わりますか？

    1. データ量の関数として、
    hatmathbfw −
    mathbfw と
    hatb − b の推定誤差をプロットせよ。ヒント: データ量は線形ではなく対数的に増やす。つまり、1000, 2000, …, 10,000 ではなく、5, 10, 20, 50, …, 10,000 とする。

    2. なぜヒントの提案が適切なのですか？

6.6 汎化

2 人の大学生が、期末試験に向けて熱心に勉強しているとする。一般に、この準備は過去年度の試験を解いて自分の実力を試すことから成る。しかし、過去の試験で良い成績を取れたからといって、本番でも優秀な成績を収められる保証はない。たとえば、Extraordinary Ellie という学生がいて、彼女の勉強は過去年度の試験問題の答えをひたすら暗記することだけだったとしよう。Ellie が並外れた記憶力を持っていたとして、これまでに見たことのあるどんな問題の答えでも完全に思い出せたとしても、これまでに見たことのない新しい問題に直面すると、彼女は途方に暮れてしまうかもしれない。これに対して、Inductive Irene という別の学生を考える。彼女の暗記能力は同程度に低いものの、パターンを見つけるのが得意だとする。もし試験が本当に前年の問題の焼き直しで構成されているなら、Ellie は Irene を難なく上回るだろう。Irene の推論したパターンが 90% の精度で予測できたとしても、100% 思い出せる Ellie には到底かなわない。しかし、試験が完全に新しい問題だけで構成されているなら、Irene は平均 90% を維持できるかもしれない。

機械学習研究者として、私たちの目標は *パターン*を発見することである。しかし、どうすれば単にデータを暗記しただけではなく、本当に一般的なパターンを発見したと確信できるのだろうか。たいていの場合、私たちの予測が役に立つのは、モデルがそのようなパターンを発見したときだけである。昨日の株価を予測したいのではなく、明日の株価を予測したいのである。すでに診断済みの患者について、すでに診断された病気を認識する必要はない。むしろ、これまで見たことのない患者における、これまで未診断の病気を認識したいのである。この問題　汎化するパターンをどう発見するか　は、機械学習の根本問題であり、おそらく統計学全体の根本問題でもある。さらに言えば、これは科学全体を包み込むより大きな問いの一部にすぎないとみなすこともできる。すなわち、個別の観測からより一般的な主張へ飛躍することが、いつ正当化されるのか、という問いである。

現実には、有限個のデータを用いてモデルを適合させなければならない。そのデータの規模は分野によって大きく異なる。多くの重要な医療問題では、利用できるデータ点は数千個程度に限られる。希少疾患を研究する場合には、数百個得られれば幸運かもしれない。対照的に、ラベル付き写真からなる最大級の公開データセット、たとえば ImageNet (Deng *et al.*, 2009) には、数百万枚の画像が含まれている。また、Flickr YFC100M データセットのようなラベルなし画像コレクションはさらに大規模で、1 億枚を超える画像を含む (Thomee *et al.*, 2016)。しかし、このような極端な規模であっても、利用可能なデータ点の数は、メガピクセル解像度で考えられるあらゆる画像の空間と比べれば、ごくわずかにすぎない。有限サンプルを扱う以上、訓練データにはうまく適合したものの、汎化可能なパターンの発見には失敗した、というリスクを常に念頭に置かなければならない。

訓練データへの適合が、背後にある真の分布への適合よりも良くなってしまう

現象を 過学習と呼び、過学習に対処する技法はしばしば 正則化手法と呼ばれる。これは統計的学習理論への適切な入門の代わりにはならないが（Boucheron *et al.* (2005), Vapnik (1998) を参照）、ここでは出発点として十分な直感を与える。本書では、さまざまな章で汎化を何度も取り上げ、さまざまなモデルにおける汎化の原理について何が分かっているかを探るとともに、実用上関心のあるタスクで（経験的に）汎化性能の向上につながることが見いだされているヒューリスティックな技法も紹介する。

6.6.1 訓練誤差と汎化誤差

標準的な教師あり学習の設定では、訓練データとテストデータは 同一の分布から独立に抽出されると仮定する。これは一般に *IID* 仮定と呼ばれる。この仮定は強いものであるが、そのような仮定がなければ、私たちはまったく立ち往生してしまうことに注意すべきである。分布 $P(X,Y)$ からサンプルされた訓練データが、異なる分布 $Q(X,Y)$ から生成されたテストデータに対する予測方法を教えてくれると、なぜ信じられるのだろうか。このような飛躍には、P と Q がどのように関係しているかについての強い仮定が必要になる。後ほど、分布の変化を許すいくつかの仮定を議論するが、まずは $P(\cdot) = Q(\cdot)$ である IID の場合を理解する必要がある。

まず、訓練誤差 R_{emp} と 汎化誤差 R を区別する必要がある。前者は訓練データセット上で計算される統計量であり、後者は基礎となる分布に関する期待値である。汎化誤差は、同じ基礎データ分布から抽出された追加のデータ例が無限に続く流れに対してモデルを適用したときに見えるものだと考えることができる。形式的には、訓練誤差は（Section 6.1 と同じ記法で）和として表される。

$$R_{\text{emp}}[\mathbf{X}, \mathbf{y}, f] = \frac{1}{n} \sum_{i=1}^{n} l(\mathbf{x}^{(i)}, y^{(i)}, f(\mathbf{x}^{(i)})), \tag{6.6.1}$$

一方、汎化誤差は積分として表される。

$$R[p, f] = E_{(\mathbf{x},y)\sim P}[l(\mathbf{x}, y, f(\mathbf{x}))] = \int \int l(\mathbf{x}, y, f(\mathbf{x}))p(\mathbf{x}, y)\, d\mathbf{x}dy. \tag{6.6.2}$$

問題なのは、汎化誤差 R を正確に計算することは決してできないということである。密度関数 $p(\mathbf{x}, y)$ の正確な形を誰も教えてはくれない。さらに、無限に続くデータ点の流れをサンプルすることもできない。したがって実際には、訓練集合から取り分けておいたランダムに選ばれた例 $\mathbf{X}'$ とラベル $\mathbf{y}'$ からなる独立なテスト集合にモデルを適用することで、汎化誤差を推定しなければならない。これは、経験的訓練誤差を計算するときに使ったのと同じ式を、テスト集合 $\mathbf{X}', \mathbf{y}'$ に対して適用することに相当する。

重要なのは、テスト集合で分類器を評価するとき、私たちは固定された分類器を扱っているということです（その分類器はテスト集合のサンプルに依存しない）。したがって、その誤差の推定は単なる平均値の推定問題である。しかし、訓練集合については同じことは言えない。最終的に得られるモデルは訓練集合の選び方に明示的に依存するため、訓練誤差は一般に、母集団における真

の誤差の偏った推定値になる。したがって汎化の中心的な問いは、訓練誤差がいつ母集団誤差（したがって汎化誤差）に近くなると期待できるか、ということである。

モデルの複雑さ

古典的理論では、単純なモデルと十分なデータがあるとき、訓練誤差と汎化誤差は近くなりがちである。しかし、より複雑なモデルや／あるいはより少ない例で作業するときには、訓練誤差は下がる一方で汎化ギャップは大きくなると予想される。これは驚くことではない。任意の n 個のデータからなるデータセットに対して、ランダムに割り当てられたラベルであっても完全に当てはめられるパラメータ集合を見つけられるほど表現力の高いモデルクラスを想像してみよ。この場合、訓練データに完全に適合したとしても、汎化誤差について何が言えるだろうか。分かっている限りでは、汎化誤差はランダム推測と大差ないかもしれない。

一般に、モデルクラスに何の制約もなければ、訓練データに適合したという事実だけから、モデルが汎化可能なパターンを発見したとは結論できない (Vapnik *et al.*, 1994)。一方で、もしモデルクラスが任意のラベルに適合できないなら、そこには何らかのパターンが発見されているはずである。モデルの複雑さに関する学習理論的な考え方は、科学哲学者カール・ポパーの考え方からいくらか着想を得ている。彼は反証可能性の基準を定式化した。ポパーによれば、あらゆる観測を説明できる理論は、そもそも科学理論ではない。結局のところ、あらゆる可能性を排除していないなら、その理論は世界について何を教えてくれるのだろうか。要するに、私たちが欲しいのは、考えうるどんな観測も説明できないはずなのに、実際に得られた観測とはたまたま整合している仮説なのである。

では、何が適切なモデル複雑度の概念を構成するのかは、複雑な問題である。しばしば、パラメータ数が多いモデルほど、より多くの任意に割り当てられたラベルに適合できる。しかし、これは必ずしも真ではない。たとえば、カーネル法は無限個のパラメータを持つ空間で動作するが、その複雑さは別の方法で制御される (Schölkopf and Smola, 2002)。しばしば有用な複雑度の概念の一つは、パラメータが取りうる値の範囲である。この観点では、パラメータが任意の値を取れることを許されたモデルの方が、より複雑だと言える。この考え方は次節で、最初の実用的な正則化手法である 重み減衰を導入するときに再び取り上げる。なお、著しく異なるモデルクラス同士（たとえば決定木とニューラルネットワーク）で複雑さを比較するのは難しいことがある。

ここで、深層ニューラルネットワークを導入するときに再び取り上げる重要な点を強調しておきる。モデルが任意のラベルに適合できる場合、訓練誤差が小さいことは必ずしも汎化誤差が小さいことを意味しない。しかし、汎化誤差が高いことを意味するわけでもない！確実に言えるのは、訓練誤差が小さいだけでは汎化誤差が小さいことの証明にはならない、ということだけである。深層ニューラルネットワークはまさにそのようなモデルである。実際にはよく汎化するが、訓練誤差だけを根拠に多くを結論づけるには強力すぎる。このような

場合には、事後的に汎化を証明するために、ホールドアウトデータにより強く依存しなければならない。ホールドアウトデータ、すなわち検証集合での誤差は、検証誤差と呼ばれる。

6.6.2 過少適合か過学習か？

訓練誤差と検証誤差を比較するときには、2 つの典型的な状況に注意する必要がある。まず、訓練誤差と検証誤差の両方が大きいのに、その差が小さい場合に注意したいである。モデルが訓練誤差を下げられないなら、それはモデルが単純すぎる（すなわち表現力が不十分である）ために、モデル化しようとしているパターンを捉えられていないのかもしれない。さらに、訓練誤差と汎化誤差の間の 汎化ギャップ（$R_{\text{emp}} - R$）が小さいので、もっと複雑なモデルを使ってもよいと考える理由がある。この現象は過少適合として知られている。

一方、上で述べたように、訓練誤差が検証誤差よりもかなり小さい場合、深刻な過学習を示している。過学習が常に悪いわけではないことに注意されたい。とくに深層学習では、最良の予測モデルがしばしばホールドアウトデータよりも訓練データではるかに良い性能を示す。最終的には、私たちが通常気にするのは汎化誤差を下げることであり、ギャップはその目的の妨げになる限りにおいてのみ問題になる。訓練誤差がゼロなら、汎化ギャップは汎化誤差そのものに等しくなり、ギャップを減らすことによってのみ前進できる。

多項式曲線当てはめ

過学習とモデル複雑度に関する古典的な直感を示すために、次のことを考える。単一の特徴量 x とそれに対応する実数値ラベル y からなる訓練データが与えられたとき、次数 d の多項式

$$\hat{y} = \sum_{i=0}^{d} x^i w_i \tag{6.6.3}$$

を見つけてラベル y を推定しようとする。これは単なる線形回帰問題であり、特徴量は x のべき乗で与えられ、モデルの重みは w_i で与えられ、バイアスは w_0 で与えられる。なぜなら、すべての x について $x^0 = 1$ だからである。これは単なる線形回帰問題なので、損失関数として二乗誤差を使える。

高次の多項式関数は低次の多項式関数よりも複雑である。なぜなら、高次多項式の方がパラメータ数が多く、モデル関数の選択範囲も広いからである。訓練データセットを固定すると、高次の多項式関数は低次の多項式よりも常に低い（少なくとも同等の）訓練誤差を達成すべきである。実際、各データ例が異なる x の値を持つなら、データ例の数に等しい次数の多項式関数は訓練集合に完全に適合できる。多項式の次数（モデル複雑度）と過少適合・過学習の関係は Fig. 6.6.1 に示す。

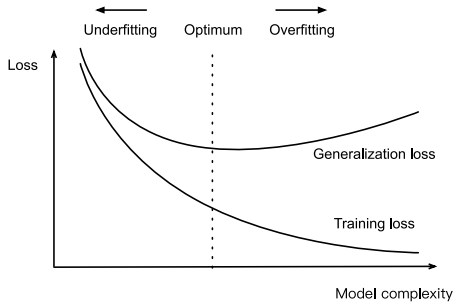


Fig. 6.6.1 モデル複雑度が過少適合と過学習に与える影響。

データセットサイズ

上の境界がすでに示しているように、もう一つ大きな考慮事項はデータセットサイズである。モデルを固定すると、訓練データセットに含まれるサンプルが少ないほど、過学習が起こりやすく（またより深刻に）なる。訓練データの量を増やすと、通常、汎化誤差は減少する。さらに一般に、データが多いことは決して損にならない。固定されたタスクとデータ分布に対しては、モデル複雑度はデータ量よりも急速に増加すべきではない。より多くのデータがあれば、より複雑なモデルを当てはめようとするかもしれない。十分なデータがなければ、単純なモデルの方が打ち負かしにくい場合がある。多くのタスクでは、深層学習が線形モデルを上回るのは、数千件以上の訓練例が利用できる場合に限られる。現在の深層学習の成功の一因は、インターネット企業、安価なストレージ、接続されたデバイス、そして経済の広範なデジタル化から生じる巨大データセットの豊富さに大きく支えられている。

6.6.3 モデル選択

通常、最終的なモデルは、さまざまな点で異なる複数のモデル（異なるアーキテクチャ、学習目的、選択した特徴量、データ前処理、学習率など）を評価した後にのみ選ぶ。多くのモデルの中から選ぶことは、適切にも モデル選択と呼ばれる。

原則として、すべてのハイパーパラメータを選び終えるまではテスト集合に触れるべきではない。モデル選択の過程でテストデータを使ってしまうと、テストデータに過学習してしまう危険がある。そうなると大変である。訓練データに過学習したとしても、テストデータでの評価があるので不正を防げる。しかし、テストデータに過学習したら、どうやってそれに気づけるだろうか。複雑さを厳密に制御できるモデルでさえ、これがいかにばかげた結果につながるかについては Ong *et al.* (2005) を参照されたい。

したがって、モデル選択にテストデータを決して頼るべきではない。とはいえ、モデルを訓練するのに使ったまさにそのデータ上で汎化誤差を推定することはできないので、モデル選択に訓練データだけを頼ることもできない。

実際の応用では、状況はもっと曖昧になる。理想的にはテストデータには一度だけ触れ、最良のモデルを評価したり、少数のモデル同士を比較したりするだけにしたいのであるが、現実のテストデータは一度使っただけで捨てられることはめったにない。実験のたびに新しいテスト集合を用意する余裕は、まずない。実際、ベンチマークデータを何十年も再利用することは、たとえば 画像分類[60] や 光学文字認識[61] のアルゴリズム発展に大きな影響を与えている。

60 

テスト集合で学習してしまう問題に対処する一般的な方法は、データを 3 分割し、訓練データセットとテストデータセットに加えて 検証集合を組み込むことである。その結果、検証データとテストデータの境界が気がかりなほど曖昧になる、という厄介な状況になる。特に断りがない限り、本書の実験では実際には、真のテスト集合ではなく、正しくは訓練データと検証データと呼ぶべきものを扱っている。したがって、本書の各実験で報告される精度は、真のテスト集合精度ではなく、実際には検証精度である。

61 

交差検証

訓練データが乏しい場合には、適切な検証集合を構成するのに十分なデータを取り分ける余裕すらないかもしれない。この問題に対する人気のある解決策の一つが、K 分割交差検証を用いることである。ここでは、元の訓練データを重ならない K 個の部分集合に分割する。その後、モデルの訓練と検証を K 回実行し、毎回 $K-1$ 個の部分集合で訓練し、別の 1 つの部分集合（その回の訓練には使わなかったもの）で検証する。最後に、K 回の実験結果を平均することで、訓練誤差と検証誤差を推定する。

6.6.4 まとめ

この節では、機械学習における汎化の基礎となる考え方をいくつか見てきた。これらの考え方のいくつかは、より深いモデルに進むと複雑で直感に反するものになる。ここでは、モデルはデータにひどく過学習しうる一方で、関連する複雑度の概念は暗黙的で直感に反することがあり（たとえば、より多くのパラメータを持つ大きなアーキテクチャの方がよりよく汎化するなど）る。最後に、いくつかの経験則を挙げておきる。

1. モデル選択には検証集合（または K 分割交差検証）を使う。
2. より複雑なモデルには、しばしばより多くのデータが必要である。
3. 関連する複雑度の概念には、パラメータ数と、それらが取りうる値の範囲の両方が含まれる。
4. 他の条件が同じなら、データが多いほど汎化はほぼ常に良くなる。
5. 汎化に関するこの議論全体は、IID 仮定を前提としている。訓練期間とテスト期間の間で分布が変化することを許すなら、さらに別の（おそらくより弱い）仮定がない限り、汎化については何も言えない。

6.6.5 演習

1. 多項式回帰の問題を正確に解けるのはいつですか。
2. 依存した確率変数があるために、問題を IID データとして扱うのが不適切になる例を少なくとも 5 つ挙げよ。
3. 訓練誤差がゼロになることはあるか。どのような状況で汎化誤差がゼロになるか。
4. K 分割交差検証の計算コストが非常に高いのはなぜですか。
5. K 分割交差検証の誤差推定値に偏りがあるのはなぜですか。
6. VC 次元は、関数クラスの関数によって任意のラベル {±1} で分類できる点の最大数として定義される。なぜこれは関数クラスの複雑さを測るのに適切でないのだろうか。ヒント：関数の大きさを考えよ。
7. 上司から、現在のアルゴリズムの性能があまり良くない難しいデータセットを渡されたとする。もっとデータが必要だと上司にどう説明するか。ヒント：データを増やすことはできないが、減らすことはできる。

6.7 重み減衰

過学習の問題を特徴づけたので、最初の正則化手法を導入できる。過学習は、より多くの訓練データを集めることで常に緩和できることを思い出そう。しかし、それにはコストがかかり、時間もかかり、あるいは完全に私たちの制御外であることもあり、短期的には不可能である。ひとまず、私たちはすでに資源が許す限り十分に高品質なデータを持っていると仮定し、データセットが与えられたものとして扱うときに利用できる手段に集中しよう。

多項式回帰の例 (Section 6.6.2) では、当てはめる多項式の次数を調整することでモデルの容量を制限できた。実際、特徴量の数を制限することは過学習を抑えるための一般的な手法である。しかし、単に特徴量を切り捨てるだけではあまりに大雑把すぎることがある。多項式回帰の例に戻って、高次元入力で何が起こりうるかを考えてみよう。多変量データへの多項式の自然な拡張は 単項式と呼ばれ、変数のべき乗の積にすぎない。単項式の次数は、べきの和である。たとえば、$x_1^2 x_2$ と $x_3 x_5^2$ はいずれも次数 3 の単項式である。

次数 d の項の数は、d が大きくなるにつれて急速に爆発的に増えることに注意されたい。k 個の変数があるとき、次数 d の単項式の数は $\binom{k-1+d}{k-1}$ である。次数を 2 から 3 に変えるといった小さな変化でも、モデルの複雑さは劇的に増加する。したがって、関数の複雑さを調整するには、よりきめ細かな手段がしばしば必要になる。

```
%matplotlib inline
from d2l import torch as d2l
import torch
from torch import nn
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
%matplotlib inline
from d2l import jax as d2l
import jax
from jax import numpy as jnp
import optax
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf
```

6.7.1 ノルムと重み減衰

パラメータの数を直接操作する代わりに、重み減衰は、パラメータが取りうる値を制限することで機能する。深層学習以外の分野では、ミニバッチ確率的勾配降下法で最適化するとき、より一般には ℓ_2 正則化と呼ばれる。重み減衰は、パラメトリック機械学習モデルを正則化するための最も広く使われている手法の一つかもしれない。この手法は、すべての関数 f の中で、関数 $f = 0$（すべての入力に値 0 を割り当てるもの）がある意味で最も単純であり、関数の複雑さはパラメータがゼロからどれだけ離れているかで測れる、という基本的な直観に動機づけられている。しかし、関数とゼロの間の距離を正確にはどのように測ればよいのだろうか？唯一の正解があるわけではない。実際、関数解析の一部やバナッハ空間の理論を含む数学の大きな分野全体が、このような問題に取り組むために存在している。

一つの単純な解釈としては、線形関数 $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ の複雑さを、その重みベクトルの何らかのノルム、たとえば $\|\mathbf{w}\|^2$ で測ることが考えられる。: numref:subsec_lin-algebra-norms で、より一般的な ℓ_p ノルムの特殊な場合である ℓ_2 ノルムと ℓ_1 ノルムを導入したことを思い出そう。小さな重みベクトルを確保する最も一般的な方法は、そのノルムを損失最小化問題に罰則項として加えることである。したがって、元の目的関数、すなわち訓練ラベルに対する予測損失を最小化することを、新しい目的関数、すなわち予測損失と罰則項の和を最小化することに置き換える。こうすると、重みベクトルが大きくなりすぎた場

合、学習アルゴリズムは訓練誤差の最小化よりも重みノルム $\|\mathbf{w}\|^2$ の最小化に注力するかもしれない。それこそが私たちの望むことである。コードで示すために、: numref:sec_linear_regression の線形回帰の例を再び取り上げる。そこでは、損失は次のように与えられていた。

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{6.7.1}$$

$\mathbf{x}^{(i)}$ は特徴量、$y^{(i)}$ はデータ例 i のラベルであり、$(\mathbf{w}, b)$ はそれぞれ重みパラメータとバイアスパラメータである。重みベクトルの大きさに罰則を与えるには、何らかの形で損失関数に $\|\mathbf{w}\|^2$ を加える必要があるが、この新しい加法的な罰則に対して、モデルは標準的な損失をどのようにトレードオフすべきだろうか？実際には、このトレードオフを 正則化定数 λ によって特徴づける。これは非負のハイパーパラメータであり、検証データを用いて調整する。

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2. \tag{6.7.2}$$

$\lambda = 0$ なら、元の損失関数に戻りる。$\lambda > 0$ なら、$\|\mathbf{w}\|$ の大きさを制限する。2 で割るのは慣習である。二次関数の微分を取るとき、2 と 1/2 が打ち消し合い、更新式が見た目にもきれいで簡潔になる。鋭い読者は、なぜ標準ノルム（すなわちユークリッド距離）ではなく、二乗したノルムを使うのかと疑問に思うかもしれない。これは計算上の都合である。ℓ_2 ノルムを二乗することで平方根が消え、重みベクトルの各成分の二乗和だけが残る。これにより、罰則項の微分を簡単に計算できる。すなわち、和の微分は微分の和に等しいのである。

さらに、そもそもなぜ ℓ_1 ノルムではなく ℓ_2 ノルムを使うのか、と疑問に思うかもしれない。実際、他の選択肢も有効であり、統計学では広く使われている。ℓ_2 正則化された線形モデルは古典的な リッジ回帰アルゴリズムを構成するが、ℓ_1 正則化された線形回帰は同様に基本的な統計手法であり、一般に ラッソ回帰として知られている。ℓ_2 ノルムを使う一つの理由は、重みベクトルの大きな成分に対して特に大きな罰則を課すことである。これにより学習アルゴリズムは、より多くの特徴量に重みを均等に分配するモデルへと偏りる。実際には、これは単一変数の測定誤差に対してより頑健にするかもしれない。対照的に、ℓ_1 罰則は、他の重みをゼロにしてしまうことで、少数の特徴量に重みを集中させるモデルを導きる。これにより、特徴選択のための有効な手法が得られ、別の理由から望ましいことがある。たとえば、モデルが少数の特徴量にしか依存しないなら、他の（捨てられた）特徴量についてデータを収集、保存、送信する必要がなくなるかもしれない。

(6.1.11) と同じ記法を用いると、ミニバッチ確率的勾配降下法による ℓ_2 正則化回帰の更新は次のようになる。

$$\mathbf{w} \leftarrow (1 - \eta\lambda)\,\mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i \in \mathcal{B}} \mathbf{x}^{(i)}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right). \tag{6.7.3}$$

これまでと同様に、推定値が観測値からどれだけずれているかに基づいて $\mathbf{w}$ を更新する。しかし同時に、$\mathbf{w}$ の大きさをゼロへ向かって縮小する。そのため、この手法はしばしば「重み減衰」と呼ばれる。罰則項だけを考えると、最適化アルゴリズムは学習の各ステップで重みを減衰させる。特徴選択とは対照的

に、重み減衰は関数の複雑さを連続的に調整する仕組みを与えてくれる。λ が小さいほど $\mathbf{w}$ への制約は弱くなり、一方で λ が大きいほど $\mathbf{w}$ はより強く制約される。対応するバイアス罰則 b^2 を含めるかどうかは実装によって異なり、ニューラルネットワークの層ごとに異なる場合もある。多くの場合、バイアス項は正則化しない。さらに、ℓ_2 正則化が他の最適化アルゴリズムでは重み減衰と等価でない場合があるとしても、重みの大きさを縮小することによる正則化という考え方自体は依然として成り立ちる。

6.7.2 高次元線形回帰

重み減衰の利点は、単純な合成例を通して示すことができる。

まず、以前と同様にデータを生成する：

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01^2). \tag{6.7.4}$$

この合成データセットでは、ラベルは入力の背後にある線形関数によって与えられ、平均 0、標準偏差 0.01 のガウスノイズによって汚されている。説明のために、問題の次元を $d = 200$ に増やし、20 例しかない小さな訓練セットで学習することで、過学習の影響を顕著にできる。

```
class Data(d2l.DataModule):
    def __init__(self, num_train, num_val, num_inputs, batch_size):
        self.save_hyperparameters()
        n = num_train + num_val
        if tab.selected('mxnet') or tab.selected('pytorch'):
            self.X = d2l.randn(n, num_inputs)
            noise = d2l.randn(n, 1) * 0.01
        if tab.selected('tensorflow'):
            self.X = d2l.normal((n, num_inputs))
            noise = d2l.normal((n, 1)) * 0.01
        if tab.selected('jax'):
            self.X = jax.random.normal(jax.random.PRNGKey(0), (n, num_inputs))
            noise = jax.random.normal(jax.random.PRNGKey(0), (n, 1)) * 0.01
        w, b = d2l.ones((num_inputs, 1)) * 0.01, 0.05
        self.y = d2l.matmul(self.X, w) + b + noise

    def get_dataloader(self, train):
        i = slice(0, self.num_train) if train else slice(self.num_train, None)
        return self.get_tensorloader([self.X, self.y], train, i)
```

6.7.3 ゼロからの実装

では、重み減衰をゼロから実装してみよう。ミニバッチ確率的勾配降下法が最適化手法なので、元の損失関数に二乗した ℓ_2 罰則を加えるだけで十分である。

ℓ_2 ノルム罰則の定義

この罰則を実装する最も便利な方法は、すべての項をその場で二乗してから和を取ることかもしれない。

```
def l2_penalty(w):
    return d2l.reduce_sum(w**2) / 2
```

モデルの定義

最終的なモデルでは、線形回帰と二乗損失は Section 6.4 から変わっていないので、d2l.LinearRegressionScratch のサブクラスを定義するだけでよいだろう。ここでの唯一の変更点は、損失に罰則項が含まれることである。

```
class WeightDecayScratch(d2l.LinearRegressionScratch):
    def __init__(self, num_inputs, lambd, lr, sigma=0.01):
        super().__init__(num_inputs, lr, sigma)
        self.save_hyperparameters()

    def loss(self, y_hat, y):
        return (super().loss(y_hat, y) +
                self.lambd * l2_penalty(self.w))
```

```
class WeightDecayScratch(d2l.LinearRegressionScratch):
    def __init__(self, num_inputs, lambd, lr, sigma=0.01):
        super().__init__(num_inputs, lr, sigma)
        self.save_hyperparameters()

    def loss(self, y_hat, y):
        return (super().loss(y_hat, y) +
                self.lambd * l2_penalty(self.w))
```

```
class WeightDecayScratch(d2l.LinearRegressionScratch):
    lambd: int = 0

    def loss(self, params, X, y, state):
        return (super().loss(params, X, y, state) +
                self.lambd * l2_penalty(params['w']))
```

```
class WeightDecayScratch(d2l.LinearRegressionScratch):
    def __init__(self, num_inputs, lambd, lr, sigma=0.01):
        super().__init__(num_inputs, lr, sigma)
        self.save_hyperparameters()

    def loss(self, y_hat, y):
        return (super().loss(y_hat, y) +
                self.lambd * l2_penalty(self.w))
```

次のコードは、20 例の訓練セットでモデルを学習し、100 例の検証セットで評価する。

```
data = Data(num_train=20, num_val=100, num_inputs=200, batch_size=5)
trainer = d2l.Trainer(max_epochs=10)

def train_scratch(lambd):
    model = WeightDecayScratch(num_inputs=200, lambd=lambd, lr=0.01)
    model.board.yscale='log'
    trainer.fit(model, data)
    if tab.selected('pytorch', 'mxnet', 'tensorflow'):
        print('L2 norm of w:', float(l2_penalty(model.w)))
    if tab.selected('jax'):
        print('L2 norm of w:',
              float(l2_penalty(trainer.state.params['w'])))
```

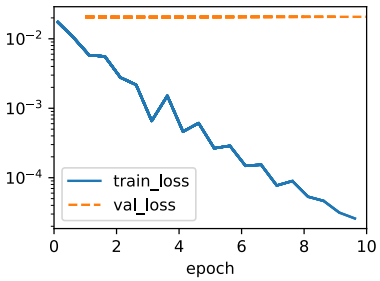
```
[07:08:03] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

正則化なしでの学習

ここでは `lambd = 0` としてこのコードを実行し、重み減衰を無効にする。訓練誤差は下がる一方で検証誤差は下がらず、ひどく過学習していることに注意されたい。これは過学習の典型例である。

```
train_scratch(0)
```

```
L2 norm of w: 0.011301761493086815
```

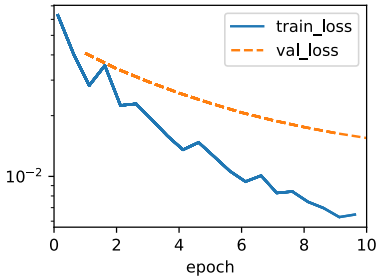


重み減衰の使用

以下では、かなり強い重み減衰をかけて実行する。訓練誤差は増加するが、検証誤差は減少することに注意されたい。これはまさに正則化から期待される効果である。

```
train_scratch(3)
```

```
L2 norm of w: 0.0016639942768961191
```



6.7.4 簡潔な実装

重み減衰はニューラルネットワーク最適化で至るところに使われているため、深層学習フレームワークでは特に便利に扱える。最適化アルゴリズム自体に重み減衰を統合し、任意の損失関数と組み合わせて簡単に使えるようにしている。さらに、この統合は計算上の利点もあり、追加の計算オーバーヘッドなしに、実装上の工夫でアルゴリズムに重み減衰を加えられる。更新の重み減衰部分は各パラメータの現在値のみに依存するため、最適化器はどうせ各パラメータに一度は触れる必要がある。

以下では、最適化器をインスタンス化するときに `weight_decay` を通じて重み減衰ハイパーパラメータを直接指定する。デフォルトでは、PyTorch は重みとバイアスの両方を同時に減衰させるが、最適化器を異なるパラメータに対して異なる方針で扱うように設定できる。ここでは、重みに対してのみ（`net.weight` パラメータに対してのみ）`weight_decay` を設定しているため、バイアス（`net.bias` パラメータ）は減衰しない。

```
class WeightDecay(d2l.LinearRegression):
    def __init__(self, wd, lr):
        super().__init__(lr)
        self.save_hyperparameters()
        self.wd = wd

    def configure_optimizers(self):
        return torch.optim.SGD([
            {'params': self.net.weight, 'weight_decay': self.wd},
            {'params': self.net.bias}], lr=self.lr)
```

```
class WeightDecay(d2l.LinearRegression):
    def __init__(self, wd, lr):
        super().__init__(lr)
        self.save_hyperparameters()
        self.wd = wd
```

(continues on next page)

(continued from previous page)

```
    def configure_optimizers(self):
        self.collect_params('.*bias').setattr('wd_mult', 0)
        return gluon.Trainer(self.collect_params(),
                             'sgd',
                             {'learning_rate': self.lr, 'wd': self.wd})
```

```
class WeightDecay(d2l.LinearRegression):
    wd: int = 0

    def configure_optimizers(self):
        # Weight Decay is not available directly within optax.sgd, but
        # optax allows chaining several transformations together
        return optax.chain(optax.additive_weight_decay(self.wd),
                           optax.sgd(self.lr))
```

```
class WeightDecay(d2l.LinearRegression):
    def __init__(self, wd, lr):
        super().__init__(lr)
        self.save_hyperparameters()
        self.net = tf.keras.layers.Dense(
            1, kernel_regularizer=tf.keras.regularizers.l2(wd),
            kernel_initializer=tf.keras.initializers.RandomNormal(0, 0.01)
        )

    def loss(self, y_hat, y):
        return super().loss(y_hat, y) + self.net.losses
```

プロットは、ゼロから重み減衰を実装したときと似ている。しかし、この版のほうが高速に動作し、実装も容易である。問題が大きくなり、作業がより日常的になるにつれて、これらの利点はさらに顕著になる。

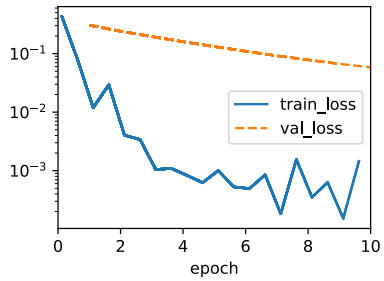
```
model = WeightDecay(wd=3, lr=0.01)
model.board.yscale='log'
trainer.fit(model, data)

if tab.selected('jax'):
    print('L2 norm of w:', float(l2_penalty(model.get_w_b(trainer.state)[0])))
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    print('L2 norm of w:', float(l2_penalty(model.get_w_b()[0])))
```

```
L2 norm of w: 0.014530565589666367
```

ここまでで、単純な線形関数を構成するものについて一つの考え方に触れた。しかし、単純な非線形関数であっても、状況ははるかに複雑になりえる。これを理解する上で有用なのが再生核ヒルベルト空間（RKHS）[62] の概念であり、これを使うことで、線形関数のために導入された道具を非線形の文脈に適用できるようになる。残念ながら、RKHS ベースのアルゴリズムは大規模で高次元のデータに対してはスケーリングがうまくいかない傾向がある。この本では、し

62 



ばしば重み減衰を深層ネットワークのすべての層に適用するという一般的なヒューリスティックを採用する。

6.7.5 まとめ

正則化は過学習に対処するための一般的な方法である。古典的な正則化手法では、学習時に損失関数へ罰則項を加えることで、学習されたモデルの複雑さを抑える。モデルを単純に保つための特定の選択肢の一つが、ℓ_2 罰則を使うことである。これにより、ミニバッチ確率的勾配降下法の更新ステップに重み減衰が現れる。実際には、重み減衰の機能は深層学習フレームワークの最適化器に備わっている。同じ訓練ループの中でも、異なるパラメータ集合に対して異なる更新挙動を持たせることができる。

6.7.6 演習

1. この節の推定問題で λ の値を変えて実験しなさい。訓練精度と検証精度を λ の関数としてプロットしなさい。何が観察できるか？
2. 検証セットを用いて λ の最適値を見つけなさい。本当に最適値だろうか？それは重要だろうか？
3. 罰則として $\|\mathbf{w}\|^2$ の代わりに $\sum_i |w_i|$ を用いた場合（ℓ_1 正則化）、更新方程式はどのようになるか？
4. $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$ であることは分かっている。行列に対しても同様の式を見つけられますか（Section 5.3.11 のフロベニウスノルムを参照）？
5. 訓練誤差と汎化誤差の関係を復習しなさい。重み減衰に加えて、訓練の増加や適切な複雑さを持つモデルの使用以外に、過学習に対処するのに役立つ方法は何だろうか？
6. ベイズ統計では、事前分布と尤度の積を用いて $P(w \mid x) \propto P(x \mid w)P(w)$ により事後分布を得る。$P(w)$ を正則化とどのように対応づけられますか？

7 分類のための線形ニューラルネットワーク

ここまでで、仕組みのすべてを学んできたので、学んだスキルをより広い種類のタスクに適用する準備が整った。分類へと話を移しても、大部分の土台は同じままである。すなわち、データを読み込み、それをモデルに通し、出力を生成し、損失を計算し、重みに関する勾配を求め、モデルを更新する。ただし、ターゲットの正確な形式、出力層のパラメータ化、そして損失関数の選択は、分類の設定に合わせて調整される。

7.1 ソフトマックス回帰

Section 6.1 では、線形回帰を導入し、Section 6.4 でスクラッチから実装し、さらに Section 6.5 では深層学習フレームワークの高水準 API を用いて面倒な計算を任せる方法も見た。

回帰は、どれくらい？や いくつ？という問いに答えたいときに私たちが手に取る定番の道具である。家がいくらで売れるかを予測したい、野球チームが何勝するかを予測したい、あるいは患者が退院するまでに何日入院するかを予測したい、といった場合には、おそらく回帰モデルを探していることになる。しかし、回帰モデルの中にも重要な違いがある。たとえば、家の価格は決して負にならず、変化はしばしば基準価格に対する相対的なものである。そのため、価格の対数に対して回帰するほうが有効な場合がある。同様に、患者が病院に滞在する日数は離散的な非負の確率変数である。したがって、最小二乗法も理想的な方法とは限らない。この種のイベント発生までの時間を扱うモデリングには、他にも多くの複雑さが伴い、それらは生存分析と呼ばれる専門分野で扱われる。

ここでのポイントは、圧倒することではなく、推定には二乗誤差を最小化するだけ以上のことがあると知ってもらうことである。さらに広く言えば、教師あり学習には回帰以上のものがたくさんある。この節では、どれくらい？ではなく どのカテゴリ？に焦点を当てる 分類問題を扱いる。

- このメールは迷惑メールフォルダに入るのか、それとも受信トレイに入るのか？
- この顧客はサブスクリプションサービスに申し込む可能性が高いのか、それとも申し込まない可能性が高いのか？
- この画像はロバ、犬、猫、それとも雄鶏を表しているのか？

- Aston は次にどの映画を最も見そうか？

- あなたは次に本のどの節を読むのか？

口語的には、機械学習の実務者は 分類という語を、微妙に異なる 2 つの問題を指して使い分けている。(i) データ例をカテゴリ（クラス）へハードに割り当てることだけに関心がある場合と、(ii) ソフトな割り当て、すなわち各カテゴリが当てはまる確率を評価したい場合である。この区別はしばしば曖昧になるが、その一因は、実際にはハードな割り当てだけが重要な場合でも、ソフトな割り当てを行うモデルを使うことが多いからである。

さらに、複数のラベルが同時に真である場合もある。たとえば、ニュース記事がエンターテインメント、ビジネス、宇宙飛行の話題を同時に扱う一方で、医療やスポーツの話題は扱わないことがある。このような記事を上のカテゴリのどれか 1 つにだけ分類するのは、あまり役に立たない。この問題は一般に マルチラベル分類[63] として知られている。概要については Tsoumakas and Katakis (2007) を、画像へのタグ付けに有効なアルゴリズムについては Huang *et al.* (2015) を参照されたい。

[63] 

7.1.1 分類

まずは手始めに、単純な画像分類問題から始めよう。ここでは、各入力は 2×2 のグレースケール画像である。各画素値を 1 つのスカラーで表せるので、4 つの特徴 x_1, x_2, x_3, x_4 が得られる。さらに、各画像は「猫」「鶏」「犬」のいずれか 1 つのカテゴリに属すると仮定する。

次に、ラベルをどう表現するかを決める必要がある。明らかな選択肢は 2 つある。最も自然な発想は、整数がそれぞれ {dog, cat, chicken} を表すように、$y \in \{1, 2, 3\}$ を選ぶことだろう。これは、そのような情報をコンピュータ上で保存するには非常に良い方法である。もしカテゴリ間に自然な順序があるなら、たとえば {baby, toddler, adolescent, young adult, adult, geriatric} を予測したい場合には、これを 順序回帰[64] 問題として扱い、この形式のままラベルを保持することにも意味があるかもしれない。さまざまなランキング損失関数の概要については Moon *et al.* (2010) を、複数のモードを持つ応答に対処するベイズ的アプローチについては Beutel *et al.* (2014) を参照されたい。

[64] 

一般に、分類問題にはクラス間の自然な順序はない。幸いなことに、統計学者たちはずっと昔にカテゴリデータを表現する簡単な方法を考案した。それが *one-hot* エンコーディングである。one-hot エンコーディングは、カテゴリ数と同じ成分数を持つベクトルである。あるデータ例のカテゴリに対応する成分を 1 にし、それ以外の成分を 0 にする。この場合、ラベル y は 3 次元ベクトルとなり、$(1, 0, 0)$ が「猫」、$(0, 1, 0)$ が「鶏」、$(0, 0, 1)$ が「犬」に対応する。

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \tag{7.1.1}$$

線形モデル

すべての可能なクラスに対応する条件付き確率を推定するには、クラスごとに 1 つ、複数の出力を持つモデルが必要である。線形モデルで分類を扱うには、出力の数だけアフィン関数が必要になる。厳密には、最後のカテゴリは他のカテゴリの和を 1 から引いたものとして決まるので 1 つ少なくてもよいのであるが、対称性の理由から、少し冗長なパラメータ化を使う。各出力はそれぞれ独自のアフィン関数に対応する。この例では、4 つの特徴と 3 つの出力カテゴリがあるので、重み（下付き文字付きの w）を表すのに 12 個のスカラー、バイアス（下付き文字付きの b）を表すのに 3 個のスカラーが必要である。すると次のようになる。

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned} \tag{7.1.2}$$

対応するニューラルネットワーク図を Fig. 7.1.1 に示す。線形回帰と同様に、単層ニューラルネットワークを使う。また、各出力 o_1, o_2, o_3 の計算はすべて入力 x_1, x_2, x_3, x_4 に依存するので、出力層は 全結合層とも呼べる。

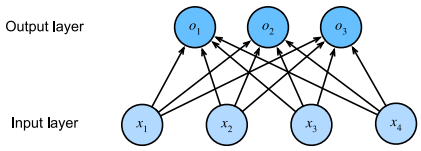


Fig. 7.1.1 Softmax regression is a single-layer neural network.

より簡潔な記法として、ベクトルと行列を使う。$\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$ のほうが、数学にもコードにもずっと適している。すべての重みを 3×4 行列にまとめ、バイアスは $\mathbf{b} \in \mathbb{R}^3$ のベクトルにまとめていることに注意されたい。

ソフトマックス

適切な損失関数があると仮定すれば、$\mathbf{o}$ とラベル $\mathbf{y}$ の差を直接最小化しようとすることもできる。分類をベクトル値回帰問題として扱うことは驚くほどうまくいく場合もあるが、それでも次の点で不十分である。

- 出力 o_i が、確率の振る舞いとして期待するように 1 に和がなる保証がない。
- 出力 o_i が非負である保証もない。たとえ出力の和が 1 になっていても、1 を超えない保証もない。

これらの点はいずれも推定問題を解きにくくし、外れ値に対して非常に脆弱な解になる。たとえば、寝室の数と家を買う可能性の間に正の線形依存があると仮定すると、豪邸を買う場合には確率が 1 を超えてしまうかもしれない！したがって、出力を「ぎゅっと押しつぶす」仕組みが必要である。

この目的を達成する方法はいくつもある。たとえば、出力 $\mathbf{o}$ は $\mathbf{y}$ の汚染された版であり、その汚染は正規分布から引いたノイズ $\boldsymbol{\epsilon}$ を加えることで起こると仮

定できる。言い換えると、$\mathbf{y} = \mathbf{o} + \boldsymbol{\epsilon}$ であり、$\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ である。これはいわゆる プロビットモデル[65] で、最初に Fechner (1860) によって導入された。魅力的ではあるが、ソフトマックスと比べると、うまく機能するわけでもなく、特に良い最適化問題にもならない。



この目的を達成する別の方法（そして非負性を保証する方法）は、指数関数 $P(y = i) \propto \exp o_i$ を使うことである。これは確かに、条件付きクラス確率が o_i の増加とともに増えるという要件を満たし、単調であり、すべての確率が非負になる。次に、各値をその和で割ることで、合計が 1 になるように変換できる。この過程を 正規化と呼ぶ。この 2 つを組み合わせると、ソフトマックス関数が得られる。

$$\hat{\mathbf{y}} = \operatorname{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}. \tag{7.1.3}$$

$\mathbf{o}$ の最大成分が、$\hat{\mathbf{y}}$ による最もありそうなクラスに対応することに注意されたい。さらに、ソフトマックス演算は引数の順序を保存するので、どのクラスに最も高い確率が割り当てられたかを調べるために、ソフトマックスを計算する必要はない。したがって、

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j. \tag{7.1.4}$$

ソフトマックスの考え方は Gibbs (1902) にさかのぼり、そこでは物理学のアイデアが取り入れられた。さらにさかのぼると、現代統計物理学の父であるボルツマンは、気体分子のエネルギー状態の分布をモデル化するためにこの手法を用いた。特に彼は、気体中の分子のような熱力学的アンサンブルにおけるあるエネルギー状態の出現頻度は、$\exp(-E/kT)$ に比例することを発見した。ここで、E は状態のエネルギー、T は温度、k はボルツマン定数である。統計学者が統計システムの「温度」を上げたり下げたりすると言うとき、それは低エネルギー状態または高エネルギー状態を優先するために T を変えることを指す。ギブスの考えに従えば、エネルギーは誤差に対応する。エネルギーベースモデル (Ranzato *et al.*, 2007) は、深層学習の問題を記述する際にこの見方を用いる。

ベクトル化

計算効率を改善するために、データのミニバッチに対して計算をベクトル化する。次元（入力数）が d の n 個のデータ例からなるミニバッチ $\mathbf{X} \in \mathbb{R}^{n \times d}$ が与えられたとする。さらに、出力カテゴリ数が q あるとする。すると、重みは $\mathbf{W} \in \mathbb{R}^{d \times q}$ を満たし、バイアスは $\mathbf{b} \in \mathbb{R}^{1 \times q}$ を満たする。

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \operatorname{softmax}(\mathbf{O}). \end{aligned} \tag{7.1.5}$$

これにより、支配的な演算は行列–行列積 $\mathbf{X}\mathbf{W}$ に高速化される。さらに、$\mathbf{X}$ の各行は 1 つのデータ例を表すので、ソフトマックス演算自体も 行ごとに計算できる。つまり、$\mathbf{O}$ の各行について、すべての要素を指数化し、その後それらを和で正規化する。ただし、大きな数に対して指数や対数を取ると数値オーバー

フローやアンダーフローを引き起こす可能性があるため、注意が必要である。深層学習フレームワークはこれを自動的に処理してくれる。

7.1.2 損失関数

特徴 $\mathbf{x}$ から確率 $\hat{\mathbf{y}}$ への写像が得られたので、この写像の精度を最適化する方法が必要である。ここでは最尤推定に頼りる。これは、Section 6.1.3 で二乗誤差損失に確率的な正当化を与えたときに出会ったのとまったく同じ方法である。

対数尤度

ソフトマックス関数はベクトル $\hat{\mathbf{y}}$ を与える。これは、任意の入力 $\mathbf{x}$ に対する各クラスの（推定された）条件付き確率、たとえば $\hat{y}_1 = P(y = \text{cat} \mid \mathbf{x})$ と解釈できる。以下では、特徴 $\mathbf{X}$ を持つデータセットに対して、ラベル $\mathbf{Y}$ は one-hot エンコーディングのラベルベクトルで表されていると仮定する。特徴が与えられたときに、実際のクラスがモデルに従ってどれだけ起こりやすいかを調べることで、推定値と現実を比較できる。

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}). \tag{7.1.6}$$

この因数分解を使えるのは、各ラベルがそれぞれの分布 $P(\mathbf{y} \mid \mathbf{x}^{(i)})$ から独立にサンプルされると仮定しているからである。項の積を最大化するのは扱いにくいので、負の対数を取って、負の対数尤度を最小化する同値な問題に変換する。

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \tag{7.1.7}$$

ここで、q クラスにわたるラベル $\mathbf{y}$ とモデル予測 $\hat{\mathbf{y}}$ の任意の組に対して、損失関数 l は

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j. \tag{7.1.8}$$

後で説明する理由により、(7.1.8) の損失関数は一般に 交差エントロピー損失と呼ばれる。$\mathbf{y}$ は長さ q の one-hot ベクトルなので、そのすべての成分 j にわたる和は、1 項を除いて消える。損失 $l(\mathbf{y}, \hat{\mathbf{y}})$ は、$\hat{\mathbf{y}}$ が確率ベクトルである限り 0 以上に下から有界である。どの成分も 1 より大きくないので、その負の対数は 0 より小さくなりえない。$l(\mathbf{y}, \hat{\mathbf{y}}) = 0$ となるのは、実際のラベルを確実に予測した場合だけである。これは、有限の重み設定では決して起こらない。なぜなら、ソフトマックス出力を 1 に近づけるには、対応する入力 o_i を無限大にする（あるいは $j \neq i$ の他のすべての出力 o_j を負の無限大にする）必要があるからである。たとえモデルが出力確率 0 を割り当てられたとしても、そのような高い確信度での誤りは無限大の損失を招く（$-\log 0 = \infty$）。

ソフトマックスと交差エントロピー損失

ソフトマックス関数と対応する交差エントロピー損失は非常に一般的なので、それらがどのように計算されるかをもう少しよく理解する価値がある。(7.1.3) を (7.1.8) の損失定義に代入し、ソフトマックスの定義を使うと、次を得る。

$$\begin{aligned} l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} \\ &= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \\ &= \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j. \end{aligned} \tag{7.1.9}$$

何が起きているのかをもう少しよく理解するために、任意のロジット o_j に関する微分を考えよう。すると、

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} - y_j = \mathrm{softmax}(\mathbf{o})_j - y_j. \tag{7.1.10}$$

言い換えると、この微分は、ソフトマックス演算で表されるモデルが割り当てた確率と、one-hot ラベルベクトルの要素で表される実際に起こったこととの差である。この意味で、これは回帰で見たものと非常によく似ている。回帰では、勾配は観測値 y と推定値 $\hat{y}$ の差でした。これは偶然ではない。指数型分布族モデルでは、対数尤度の勾配はまさにこの項で与えられる。この事実により、実際の勾配計算は容易になる。

次に、単一の結果だけでなく、結果の分布全体を観測する場合を考える。ラベル $\mathbf{y}$ には先ほどと同じ表現を使える。違いは、$(0, 0, 1)$ のような二値成分だけを含むベクトルではなく、$(0.1, 0.2, 0.7)$ のような一般的な確率ベクトルになることである。(7.1.8) で損失 l を定義するために以前使った数学は、この場合でもそのままうまく機能する。ただし、解釈が少し一般的になるだけである。それは、ラベル分布に対する損失の期待値である。この損失は 交差エントロピー損失と呼ばれ、分類問題で最もよく使われる損失の 1 つである。情報理論の基本を少し導入することで、この名前をわかりやすくできる。要するに、これは、私たちが観測するもの $\mathbf{y}$ を、予測した結果 $\hat{\mathbf{y}}$ に対して符号化するのに必要なビット数を測るものである。以下でごく基本的な説明をする。情報理論の詳細については Cover and Thomas (1999) または MacKay (2003) を参照されたい。

7.1.3 情報理論の基本

多くの深層学習論文では、情報理論の直感や用語が使われる。それらを理解するには、共通の言語が必要である。これは生存ガイドである。情報理論は、情報（データとも呼ばれる）の符号化、復号、伝送、操作の問題を扱いる。

エントロピー

情報理論の中心的な考え方は、データに含まれる情報量を定量化することである。これは、データ圧縮の限界を与える。分布 P に対する エントロピー $H[P]$ は次のように定義される。

$$H[P] = \sum_j -P(j) \log P(j). \tag{7.1.11}$$

情報理論の基本定理の 1 つは、分布 P からランダムに生成されたデータを符号化するには、少なくとも $H[P]$ 「ナット」が必要だと述べている (Shannon, 1948)。「ナット」とは何か気になるかもしれないが、これはビットの e を底とする符号化における対応物である。つまり、1 ナットは $\frac{1}{\log(2)} \approx 1.44$ ビットである。

驚き

圧縮と予測にどんな関係があるのか、不思議に思うかもしれない。圧縮したいデータストリームがあると想像してみよう。次のトークンをいつも簡単に予測できるなら、そのデータは圧縮しやすいである。極端な例として、ストリーム中のすべてのトークンが常に同じ値を取る場合を考えよう。それはとても退屈なデータストリームです！しかも退屈なだけでなく、予測も簡単である。トークンが常に同じなので、ストリームの内容を伝えるために何の情報も送る必要がない。予測しやすいものは、圧縮しやすいのである。

しかし、すべての事象を完全には予測できないなら、ときには驚くことがある。ある事象に割り当てられた確率が低いほど、私たちの驚きは大きくなる。クロード・シャノンは、事象 j を観測したときの 驚きを定量化するために、$\log \frac{1}{P(j)} = -\log P(j)$ を採用した。ここで、事象 j に（主観的に）確率 $P(j)$ を割り当てていたとする。(7.1.11) で定義したエントロピーは、データ生成過程に本当に一致する正しい確率を割り当てたときの 期待驚きである。

交差エントロピーの再考

では、エントロピーが真の確率を知っている人が経験する驚きの度合いだとすると、交差エントロピーとは何だろうか？P から Q への交差エントロピーを $H(P,Q)$ と書くと、これは、実際には確率 P に従って生成されたデータを見たときに、主観的確率 Q を持つ観測者が感じる期待驚きである。これは $H(P,Q) \stackrel{\text{def}}{=} \sum_j -P(j) \log Q(j)$ で与えられる。最小の交差エントロピーは $P = Q$ のときに達成される。この場合、P から Q への交差エントロピーは $H(P,P) = H(P)$ である。

要するに、交差エントロピーによる分類目的関数は 2 通りに考えられる。(i) 観測データの尤度を最大化すること、(ii) ラベルを伝えるのに必要な驚き（したがってビット数）を最小化することである。

7.1.4 要約と考察

この節では、離散的な出力空間上で最適化を可能にする、最初の非自明な損失関数に出会った。その設計の鍵は、離散カテゴリを確率分布からのサンプルとして扱うという確率論的アプローチを取ったことでした。その副作用として、ソフトマックスという便利な活性化関数にも出会った。これは通常のニューラルネットワーク層の出力を、有効な離散確率分布へ変換する。ソフトマックスと組み合わせた交差エントロピー損失の微分は、二乗誤差の微分と非常によく似ていることも見た。すなわち、期待される振る舞いとその予測との差を取るという形である。そして、その表面をかすめただけではあったが、統計物理学や情報理論との刺激的なつながりにも触れた。

これで先へ進むには十分であり、できれば興味をそそるには十分であるが、ここではまだ深く掘り下げたとは言えない。とりわけ、計算上の考慮事項を飛ばした。具体的には、d 個の入力と q 個の出力を持つ全結合層では、パラメータ化と計算コストは $O(dq)$ であり、実用上は非常に高くつくことがある。幸いなことに、d 個の入力を q 個の出力に変換するこのコストは、近似と圧縮によって削減できる。たとえば Deep Fried Convnets (Yang *et al.*, 2015) は、置換、フーリエ変換、スケーリングを組み合わせて、コストを二次から対数線形へ削減する。同様の技法は、より高度な構造化行列近似にも有効です (Sindhwani *et al.*, 2015)。最後に、圧縮係数 n に基づいて、少しの精度と引き換えに計算コストと保存コストを削減することを受け入れるなら、四元数風の分解を用いてコストを $O(\frac{dq}{n})$ にまで下げることもできる (Zhang *et al.*, 2021)。これは活発な研究分野である。難しいのは、必ずしも最もコンパクトな表現や最小の浮動小数点演算数を目指すのではなく、現代の GPU 上で最も効率よく実行できる解を目指すからである。

7.1.5 演習

1. 指数型分布族とソフトマックスの関係をさらに深く調べることができる。
   1. ソフトマックスに対する交差エントロピー損失 $l(\mathbf{y}, \hat{\mathbf{y}})$ の 2 階微分を計算せよ。
   2. $\mathrm{softmax}(\mathbf{o})$ によって与えられる分布の分散を計算し、それが上で計算した 2 階微分と一致することを示せ。
2. 3 つのクラスがあり、それらが等確率で現れる、すなわち確率ベクトルが $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ であると仮定する。
   1. これに対して 2 進コードを設計しようとすると何が問題か。
   2. より良いコードを設計できるか？ヒント：2 つの独立な観測を符号化しようとすると何が起こるか？n 個の観測をまとめて符号化するとどうか？
3. 物理的なワイヤを通じて送られる信号を符号化するとき、エンジニアは必ずしも 2 進コードを使うわけではない。たとえば、PAM-3[66] は、2 つの信号レベル $\{0, 1\}$ の代わりに 3 つの信号レベル $\{-1, 0, 1\}$ を使う。整数の範囲 $\{0, \ldots, 7\}$ を送るには、何個の 3 進単位が必要であるか？電子工学の観点から、なぜこれがより良い考えなのか？

66 

4. Bradley–Terry モデル[67] は、選好を表すためにロジスティックモデルを使う。ユーザがリンゴとオレンジのどちらかを選ぶとき、スコア o_{apple} と o_{orange} を仮定する。要件は、スコアが大きいほど対応する項目が選ばれる確率が高くなり、最大スコアの項目が最も選ばれやすいことです (Bradley and Terry, 1952)。

   1. ソフトマックスがこの要件を満たすことを証明せよ。

   2. リンゴでもオレンジでもないものを選ぶというデフォルトの選択肢を許したい場合はどうなるか？ヒント：今やユーザには 3 つの選択肢がある。

5. ソフトマックスは次の写像に由来してその名を得ている：$\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$。

   1. $\text{RealSoftMax}(a, b) > \max(a, b)$ を証明せよ。

   2. 両者の差をどこまで小さくできるか？ヒント：一般性を失わずに $b = 0$ かつ $a \geq b$ としてよい。

   3. $\lambda > 0$ のとき、$\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b)$ に対してもこれが成り立つことを証明せよ。

   4. $\lambda \to \infty$ で $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$ となることを示せ。

   5. 対応する softmin 関数を構成せよ。

   6. これを 2 つより多い数に拡張せよ。

6. 関数 $g(\mathbf{x}) \stackrel{\text{def}}{=} \log \sum_i \exp x_i$ は、対数分配関数[68] とも呼ばれる。

   1. この関数が凸であることを証明せよ。ヒント：そのために、1 階微分がソフトマックス関数の確率に等しいことを使い、2 階微分が分散であることを示せ。

   2. g が平行移動不変、すなわち $g(\mathbf{x} + b) = g(\mathbf{x})$ であることを示せ。

   3. いくつかの座標 x_i が非常に大きい場合はどうなるか？すべてが非常に小さい場合はどうか？

   4. $b = \max_i x_i$ を選ぶと、数値的に安定な実装になることを示せ。

7. ある確率分布 P があると仮定する。$\alpha > 0$ に対して $Q(i) \propto P(i)^\alpha$ となる別の分布 Q を選ぶとする。

   1. どの α が温度を 2 倍にすることに対応するか？どの選択が温度を半分にすることに対応するか？

   2. 温度を 0 に近づけると何が起こるか？

   3. 温度を ∞ に近づけると何が起こるか？

7.2 画像分類データセット

画像分類で広く使われているデータセットの 1 つに、手書き数字の MNIST データセット[69] (LeCun *et al.*, 1998) がある。1990 年代に公開された当時、これはほとんどの機械学習アルゴリズムにとって手強い課題であり、28×28 ピクセル解像度の画像 60,000 枚（加えて 10,000 枚のテストデータセット）から構成されていた。参考までに言うと、1995 年当時、64MB という膨大な RAM と 5 MFLOPs という驚異的な性能を備えた Sun SPARCStation 5 は、AT&T Bell Laboratories における機械学習向けの最先端機器と見なされていた。数字認識で高い精度を達成することは、1990 年代に USPS の郵便物仕分けを自動化するうえで重要だった。LeNet-5 (LeCun *et al.*, 1995) のような深層ネットワーク、変換不変性を組み込んだサポートベクターマシン (Schölkopf *et al.*, 1996)、接線距離分類器 (Simard *et al.*, 1998) は、いずれも 1% 未満の誤り率を達成できた。

[69] 

10 年以上にわたり、MNIST は機械学習アルゴリズムを比較するための基準として機能してきた。ベンチマークデータセットとして長く活躍したが、今日の基準では単純なモデルでも 95% を超える分類精度を達成できるため、強いモデルと弱いモデルを見分けるには不十分である。さらに、このデータセットでは、通常の多くの分類問題では見られないほど非常に高い精度が可能である。そのため、アルゴリズムの発展は、クリーンなデータセットの利点を活かせる特定の系統、たとえばアクティブセット法や境界探索型アクティブセットアルゴリズムへと偏っていった。今日では、MNIST はベンチマークというよりも、むしろ健全性確認のためのチェックとして使われている。ImageNet (Deng *et al.*, 2009) のほうが、はるかに現実的な課題を提示す。残念ながら、ImageNet は大きすぎるため、この本の多くの例や図示には向いていない。というのも、例を対話的に動かせるようにするには、学習に時間がかかりすぎるからである。代わりとして、以降の節では、質的には似ているもののはるかに小さい Fashion-MNIST データセット (Xiao *et al.*, 2017) を扱いる。これは 28×28 ピクセル解像度の 10 種類の衣類画像を含んでいる。

```
%matplotlib inline
import time
from d2l import torch as d2l
import torch
import torchvision
from torchvision import transforms

d2l.use_svg_display()
```

```
%matplotlib inline
import time
from d2l import mxnet as d2l
from mxnet import gluon, npx
from mxnet.gluon.data.vision import transforms
npx.set_np()
```

(continues on next page)

(continued from previous page)

```
d2l.use_svg_display()
```

```
%matplotlib inline
from d2l import jax as d2l
import jax
from jax import numpy as jnp
import numpy as np
import time
import tensorflow as tf
import tensorflow_datasets as tfds

d2l.use_svg_display()
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
%matplotlib inline
import time
from d2l import tensorflow as d2l
import tensorflow as tf

d2l.use_svg_display()
```

7.2.1 データセットの読み込み

Fashion-MNIST データセットは非常に有用なので、主要なフレームワークはすべて、その前処理済みバージョンを提供している。組み込みのフレームワーク用ユーティリティを使って、ダウンロードしてメモリに読み込むことができる。

```
class FashionMNIST(d2l.DataModule):  #@save
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor()])
        self.train = torchvision.datasets.FashionMNIST(
            root=self.root, train=True, transform=trans, download=True)
        self.val = torchvision.datasets.FashionMNIST(
            root=self.root, train=False, transform=trans, download=True)
```

```
class FashionMNIST(d2l.DataModule):  #@save
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor()])
        self.train = gluon.data.vision.FashionMNIST(
            train=True).transform_first(trans)
        self.val = gluon.data.vision.FashionMNIST(
            train=False).transform_first(trans)
```

```
class FashionMNIST(d2l.DataModule):  #@save
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        self.train, self.val = tf.keras.datasets.fashion_mnist.load_data()
```

```
class FashionMNIST(d2l.DataModule):  #@save
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        self.train, self.val = tf.keras.datasets.fashion_mnist.load_data()
```

Fashion-MNIST は 10 カテゴリの画像からなり、各カテゴリにつき学習データセットには 6000 枚、テストデータセットには 1000 枚が含まれている。テストデータセットはモデル性能を評価するために使われる（学習には使ってはいけない）。したがって、学習セットとテストセットにはそれぞれ 60,000 枚と 10,000 枚の画像が含まれている。

```
data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

```
(60000, 10000)
```

```
data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

```
Downloading /opt/mxnet/datasets/fashion-mnist/train-images-idx3-ubyte.gz from␣
↪https://apache-mxnet.s3-accelerate.dualstack.amazonaws.com/gluon/dataset/
↪fashion-mnist/train-images-idx3-ubyte.gz...
Downloading /opt/mxnet/datasets/fashion-mnist/train-labels-idx1-ubyte.gz from␣
↪https://apache-mxnet.s3-accelerate.dualstack.amazonaws.com/gluon/dataset/
↪fashion-mnist/train-labels-idx1-ubyte.gz...
[07:00:13] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
Downloading /opt/mxnet/datasets/fashion-mnist/t10k-images-idx3-ubyte.gz from␣
↪https://apache-mxnet.s3-accelerate.dualstack.amazonaws.com/gluon/dataset/
↪fashion-mnist/t10k-images-idx3-ubyte.gz...
```

(continues on next page)

(continued from previous page)

```
Downloading /opt/mxnet/datasets/fashion-mnist/t10k-labels-idx1-ubyte.gz from␣
↪https://apache-mxnet.s3-accelerate.dualstack.amazonaws.com/gluon/dataset/
↪fashion-mnist/t10k-labels-idx1-ubyte.gz...
```

```
(60000, 10000)
```

```
data = FashionMNIST(resize=(32, 32))
len(data.train[0]), len(data.val[0])
```

```
(60000, 10000)
```

```
data = FashionMNIST(resize=(32, 32))
len(data.train[0]), len(data.val[0])
```

```
(60000, 10000)
```

上の画像はグレースケールで、32×32 ピクセル解像度に拡大されている。これは、元の MNIST データセットが（2 値の）白黒画像で構成されていたのと似ている。ただし、現代の画像データの多くは 3 チャネル（赤、緑、青）を持ち、ハイパースペクトル画像では 100 チャネルを超えることもある（HyMap センサーは 126 チャネルを持つ）。慣例として、画像は $c \times h \times w$ のテンソルとして保存する。ここで c は色チャネル数、h は高さ、w は幅である。

```
data.train[0][0].shape
```

```
torch.Size([1, 32, 32])
```

データセットを可視化するためのユーティリティ関数を定義する。

Fashion-MNIST のカテゴリには、人間にとって理解しやすい名前が付いている。次の便利メソッドは、数値ラベルとその名前を相互に変換する。

```
@d2l.add_to_class(FashionMNIST)  #@save
def text_labels(self, indices):
    """Return text labels."""
    labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

7.2.2 ミニバッチの読み込み

学習セットとテストセットから読み込む際の手間を減らすために、ゼロから作るのではなく、組み込みのデータイテレータを使う。繰り返しになるが、各反

復でデータイテレータはサイズ batch_size のミニバッチを読み込みる。また、学習データのイテレータではデータ例をランダムにシャッフルする。

```
@d2l.add_to_class(FashionMNIST)  #@save
def get_dataloader(self, train):
    data = self.train if train else self.val
    return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
                                       num_workers=self.num_workers)
```

```
@d2l.add_to_class(FashionMNIST)  #@save
def get_dataloader(self, train):
    data = self.train if train else self.val
    return gluon.data.DataLoader(data, self.batch_size, shuffle=train,
                                 num_workers=self.num_workers)
```

```
@d2l.add_to_class(FashionMNIST)  #@save
def get_dataloader(self, train):
    data = self.train if train else self.val
    process = lambda X, y: (tf.expand_dims(X, axis=3) / 255,
                            tf.cast(y, dtype='int32'))
    resize_fn = lambda X, y: (tf.image.resize_with_pad(X, *self.resize), y)
    shuffle_buf = len(data[0]) if train else 1
    if tab.selected('tensorflow'):
        return tf.data.Dataset.from_tensor_slices(process(*data)).batch(
            self.batch_size).map(resize_fn).shuffle(shuffle_buf)
    if tab.selected('jax'):
        return tfds.as_numpy(
            tf.data.Dataset.from_tensor_slices(process(*data)).batch(
                self.batch_size).map(resize_fn).shuffle(shuffle_buf))
```

```
@d2l.add_to_class(FashionMNIST)  #@save
def get_dataloader(self, train):
    data = self.train if train else self.val
    process = lambda X, y: (tf.expand_dims(X, axis=3) / 255,
                            tf.cast(y, dtype='int32'))
    resize_fn = lambda X, y: (tf.image.resize_with_pad(X, *self.resize), y)
    shuffle_buf = len(data[0]) if train else 1
    if tab.selected('tensorflow'):
        return tf.data.Dataset.from_tensor_slices(process(*data)).batch(
            self.batch_size).map(resize_fn).shuffle(shuffle_buf)
    if tab.selected('jax'):
        return tfds.as_numpy(
            tf.data.Dataset.from_tensor_slices(process(*data)).batch(
                self.batch_size).map(resize_fn).shuffle(shuffle_buf))
```

これがどのように動くかを見るために、train_dataloader メソッドを呼び出して画像のミニバッチを読み込んでみよう。そこには 64 枚の画像が含まれている。

```
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64
```

画像の読み込みにかかる時間を見てみよう。組み込みローダーではあるが、ものすごく高速というわけではない。それでも、深層ネットワークで画像を処理するほうがかなり時間がかかるので、これで十分である。したがって、ネットワークの学習が I/O に制約されないことが重要である。

```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

```
'3.69 sec'
```

7.2.3 可視化

これから Fashion-MNIST データセットを頻繁に使う。便利関数 show_images を使えば、画像とそれに対応するラベルを可視化できる。実装の詳細は省略し、ここではインターフェースだけを示す。こうしたユーティリティ関数については、どのように動くかではなく、d2l.show_images をどう呼び出すかだけを知っていれば十分である。

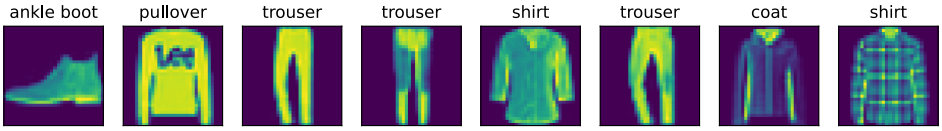
```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """Plot a list of images."""
    raise NotImplementedError
```

実際に使ってみよう。一般に、学習に使うデータを可視化して確認するのは良い考えである。人間は異常を見つけるのが非常に得意なので、可視化は実験設計におけるミスや誤りに対する追加の安全策として役立ちる。以下は、学習データセットの最初のいくつかのデータ例について、画像とそれに対応するラベル（テキスト表示）である。

```
@d2l.add_to_class(FashionMNIST)  #@save
def visualize(self, batch, nrows=1, ncols=8, labels=[]):
    X, y = batch
    if not labels:
        labels = self.text_labels(y)
    if tab.selected('mxnet', 'pytorch'):
        d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels)
    if tab.selected('tensorflow'):
        d2l.show_images(tf.squeeze(X), nrows, ncols, titles=labels)
    if tab.selected('jax'):
        d2l.show_images(jnp.squeeze(X), nrows, ncols, titles=labels)

batch = next(iter(data.val_dataloader()))
data.visualize(batch)
```

これで、以降の節で Fashion-MNIST データセットを扱う準備が整った。



7.2.4 まとめ

これで、分類に使う少し現実的なデータセットが手に入った。Fashion-MNIST は、10 カテゴリを表す画像からなる衣類分類データセットである。このデータセットは、単純な線形モデルから高度な残差ネットワークまで、さまざまなネットワーク設計を評価するために、以降の節や章で使う。画像を扱うときによく行うように、画像は（バッチサイズ、チャネル数、高さ、幅）という形状のテンソルとして読み込む。今のところ画像はグレースケールなのでチャネルは 1 つだけである（上の可視化では見やすさのために疑似カラーのパレットを使っている）。

最後に、データイテレータは効率的な性能を実現するうえで重要な要素である。たとえば、画像の解凍、動画のトランスコード、その他の前処理を効率よく行うために GPU を使うこともある。可能な限り、高性能計算を活用するよく実装されたデータイテレータに頼るべきである。そうすることで、学習ループが遅くなるのを避けられる。

7.2.5 演習

1. `batch_size` を小さくする（たとえば 1 にする）と、読み込み性能に影響するか？
2. データイテレータの性能は重要である。現在の実装は十分に高速だと思うか？改善するためのさまざまな方法を調べよ。ボトルネックがどこにあるかを調べるために、システムプロファイラを使っよ。
3. フレームワークのオンライン API ドキュメントを確認せよ。ほかにどのようなデータセットがあるか？

7.3 分類の基礎モデル

回帰の場合、スクラッチ実装とフレームワークの機能を用いた簡潔な実装がかなり似ていることに気づいたかもしれない。分類でも同じことが言える。この本の多くのモデルは分類を扱うため、この設定を特に支援する機能を追加しておく価値がある。この節では、今後のコードを簡潔にするために、分類モデル用の基底クラスを提供する。

```
from d2l import torch as d2l
import torch
```

```
from d2l import mxnet as d2l
from mxnet import autograd, np, npx, gluon
npx.set_np()
```

```
from d2l import jax as d2l
from functools import partial
from jax import numpy as jnp
import jax
import optax
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

7.3.1 Classifier クラス

以下で Classifier クラスを定義する。validation_step では、検証バッチに対する損失値と分類精度の両方を報告する。num_val_batches バッチごとに 1 回更新を描画する。これにより、検証データ全体に対する平均損失と精度を生成できるという利点がある。最後のバッチの例数が少ない場合、これらの平均値は厳密には正しくないが、コードを簡潔に保つためにこの小さな違いは無視する。

```
class Classifier(d2l.Module):  #@save
    """ 分類モデルの基底クラス。"""
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)
        self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False)
```

```
class Classifier(d2l.Module):  #@save
    """ 分類モデルの基底クラス。"""
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)
        self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False)
```

```
class Classifier(d2l.Module):  #@save
    """ 分類モデルの基底クラス。"""
    def training_step(self, params, batch, state):
```

(continues on next page)

(continued from previous page)

```
        # ここで value はタプルである。BatchNorm 層を持つモデルでは
        # 損失が補助データを返す必要があるため
        value, grads = jax.value_and_grad(
            self.loss, has_aux=True)(params, batch[:-1], batch[-1], state)
        l, _ = value
        self.plot("loss", l, train=True)
        return value, grads

    def validation_step(self, params, batch, state):
        # 2 つ目の戻り値は破棄する。これは BatchNorm 層を持つモデルの
        # 学習に使われる。損失も補助データを返すためである
        l, _ = self.loss(params, batch[:-1], batch[-1], state)
        self.plot('loss', l, train=False)
        self.plot('acc', self.accuracy(params, batch[:-1], batch[-1], state),
                  train=False)
```

```
class Classifier(d2l.Module):  #@save
    """ 分類モデルの基底クラス。"""
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)
        self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False)
```

デフォルトでは、線形回帰の文脈で行ったのと同様に、ミニバッチ上で動作する確率的勾配降下法オプティマイザを使う。

```
@d2l.add_to_class(d2l.Module)  #@save
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

```
@d2l.add_to_class(d2l.Module)  #@save
def configure_optimizers(self):
    params = self.parameters()
    if isinstance(params, list):
        return d2l.SGD(params, self.lr)
    return gluon.Trainer(params, 'sgd', {'learning_rate': self.lr})
```

```
@d2l.add_to_class(d2l.Module)  #@save
def configure_optimizers(self):
    return optax.sgd(self.lr)
```

```
@d2l.add_to_class(d2l.Module)  #@save
def configure_optimizers(self):
    return tf.keras.optimizers.SGD(self.lr)
```

7.3.2 精度

予測された確率分布 y_hat が与えられたとき、ハードな予測を出力しなければならない場合には、通常、予測確率が最も高いクラスを選ぶ。実際、多くのアプリケーションでは何らかの選択を行う必要がある。たとえば Gmail は、メールを“Primary”、“Social”、“Updates”、“Forums”、または“Spam”に分類しなければならない。内部的には確率を推定しているかもしれないが、最終的にはクラスの中から 1 つを選ばなければならない。

予測がラベルクラス y と一致しているとき、それらは正解である。分類精度は、すべての予測のうち正しいものの割合である。精度を直接最適化するのは難しい場合があるが（微分不可能であるため）、しばしば私たちが最も重視する性能指標である。ベンチマークにおいては、しばしば 最も重視される指標である。そのため、分類器を学習するときにはほぼ常に精度を報告する。

精度は次のように計算する。まず、y_hat が行列である場合、2 次元目に各クラスの予測スコアが格納されていると仮定する。argmax を用いて、各行で最大の要素のインデックスから予測クラスを得る。次に、予測クラスと正解 y を要素ごとに比較する。等価演算子 == はデータ型に敏感なので、y_hat のデータ型を y に合わせて変換する。その結果は、0（偽）と 1（真）の要素を含むテンソルになる。それらを合計すると、正しい予測の数が得られる。

```
@d2l.add_to_class(Classifier)  #@save
def accuracy(self, Y_hat, Y, averaged=True):
    """ 正しい予測の数を計算する。"""
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    preds = d2l.astype(d2l.argmax(Y_hat, axis=1), Y.dtype)
    compare = d2l.astype(preds == d2l.reshape(Y, -1), d2l.float32)
    return d2l.reduce_mean(compare) if averaged else compare
```

```
@d2l.add_to_class(Classifier)  #@save
def accuracy(self, Y_hat, Y, averaged=True):
    """ 正しい予測の数を計算する。"""
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    preds = d2l.astype(d2l.argmax(Y_hat, axis=1), Y.dtype)
    compare = d2l.astype(preds == d2l.reshape(Y, -1), d2l.float32)
    return d2l.reduce_mean(compare) if averaged else compare
```

```
@d2l.add_to_class(Classifier)  #@save
@partial(jax.jit, static_argnums=(0, 5))
def accuracy(self, params, X, Y, state, averaged=True):
    """ 正しい予測の数を計算する。"""
    Y_hat = state.apply_fn({'params': params,
                            'batch_stats': state.batch_stats},  # BatchNorm の
み
                           *X)
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    preds = d2l.astype(d2l.argmax(Y_hat, axis=1), Y.dtype)
    compare = d2l.astype(preds == d2l.reshape(Y, -1), d2l.float32)
    return d2l.reduce_mean(compare) if averaged else compare
```

```
@d2l.add_to_class(Classifier)  #@save
def accuracy(self, Y_hat, Y, averaged=True):
    """ 正しい予測の数を計算する。"""
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    preds = d2l.astype(d2l.argmax(Y_hat, axis=1), Y.dtype)
    compare = d2l.astype(preds == d2l.reshape(Y, -1), d2l.float32)
    return d2l.reduce_mean(compare) if averaged else compare
```

7.3.3 要約

分類は十分に一般的な問題であるため、専用の便利関数を用意する価値がある。分類において中心的に重要なのは、分類器の 精度である。しばしば私たちが主に気にするのは精度であるが、統計的・計算的な理由から、分類器はさまざまな他の目的を最適化するように学習される。しかし、学習中にどの損失関数を最小化したとしても、分類器の精度を経験的に評価するための便利なメソッドがあると有用である。

7.3.4 演習

1. 検証損失を L_v とし、この節の損失関数による平均化で計算される、その簡便だが粗い推定値を L_v^q とする。最後に、最後のミニバッチ上の損失を l_v^b とする。L_v を L_v^q、l_v^b、およびサンプルサイズとミニバッチサイズで表せ。
2. この簡便だが粗い推定値 L_v^q が不偏であることを示せ。すなわち、$E[L_\mathrm{v}] = E[L_\mathrm{v}^\mathrm{q}]$ を示せ。それでもなお、なぜ L_v を使いたいのか。
3. 多クラス分類の損失が与えられたとき、y を見たときに y' を推定する際の罰則を $l(y, y')$ とし、確率 $p(y \mid x)$ が与えられているとする。y' の最適な選択の規則を定式化せよ。ヒント: l と $p(y \mid x)$ を用いて期待損失を表せ。

7.4 スクラッチからのソフトマックス回帰の実装

ソフトマックス回帰は非常に基本的な手法なので、ぜひ自分で実装できるようになっておくべきだと考える。ここでは、モデルのソフトマックス固有の部分の定義に限定し、線形回帰の節で使った他の構成要素、たとえば学習ループなどは再利用する。

```
from d2l import torch as d2l
import torch
```

```
from d2l import mxnet as d2l
from mxnet import autograd, np, npx, gluon
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
from functools import partial
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

7.4.1 ソフトマックス

まず最も重要な部分から始めよう。それは、スカラーを確率へ写像することである。復習として、Section 5.3.6 および Section 5.3.7 で説明したように、テンソルの特定の次元に沿った和演算を思い出そう。行列 `X` に対して、すべての要素の和を計算する（デフォルト）ことも、特定の軸に沿った要素の和だけを計算することもできる。`axis` 変数を使うと、行方向および列方向の和を計算できる。

```
X = d2l.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
d2l.reduce_sum(X, 0, keepdims=True), d2l.reduce_sum(X, 1, keepdims=True)
```

```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

ソフトマックスの計算には 3 つの手順がある。(i) 各要素の指数を取る。(ii) 各行について和を取り、各サンプルの正規化定数を計算する。(iii) 各行をその正規化定数で割り、結果の和が 1 になるようにする。

$$\mathrm{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}. \tag{7.4.1}$$

分母の（対数を取ったもの）は（対数）分配関数と呼ばれる。これは熱力学的アンサンブルにおけるすべての可能な状態を総和するために、統計物理学[70] で導入された。実装は簡単である。

[70] 

```
def softmax(X):
    X_exp = d2l.exp(X)
    partition = d2l.reduce_sum(X_exp, 1, keepdims=True)
    return X_exp / partition  # ここではブロードキャスト機構が適用される
```

任意の入力 `X` に対して、各要素を非負の数に変換する。各行の和は 1 になる。これは確率として必要な性質である。注意: 上のコードは、非常に大きい値や非

常に小さい値に対しては頑健ではない。何が起きているかを示すには十分であるが、実用上の目的でこのコードをそのまま使うべきではない。深層学習フレームワークにはこのような保護機構が組み込まれており、今後は組み込みのソフトマックスを使う。

```
X = d2l.rand((2, 5))
X_prob = softmax(X)
X_prob, d2l.reduce_sum(X_prob, 1)
```

```
(tensor([[0.1481, 0.2050, 0.2577, 0.1232, 0.2661],
         [0.2167, 0.2010, 0.2468, 0.1128, 0.2227]]),
 tensor([1.0000, 1.0000]))
```

```
X = d2l.rand(2, 5)
X_prob = softmax(X)
X_prob, d2l.reduce_sum(X_prob, 1)
```

```
(array([[0.17777154, 0.1857739 , 0.20995119, 0.23887765, 0.18762572],
        [0.24042214, 0.1757977 , 0.23786479, 0.15572716, 0.19018826]]),
 array([1., 1.]))
```

```
X = jax.random.uniform(jax.random.PRNGKey(d2l.get_seed()), (2, 5))
X_prob = softmax(X)
X_prob, d2l.reduce_sum(X_prob, 1)
```

```
(Array([[0.16229036, 0.13327661, 0.22257593, 0.1909562 , 0.2909009 ],
        [0.30952108, 0.17390645, 0.18135622, 0.19830671, 0.13690951]],      ␣
↪dtype=float32),
 Array([1., 1.], dtype=float32))
```

```
X = d2l.rand((2, 5))
X_prob = softmax(X)
X_prob, d2l.reduce_sum(X_prob, 1)
```

```
(<tf.Tensor: shape=(2, 5), dtype=float32, numpy=
 array([[0.30452266, 0.15993236, 0.17174596, 0.160699  , 0.20310006],
        [0.24015403, 0.09632394, 0.21000664, 0.25052655, 0.20298874]],
       dtype=float32)>,
 <tf.Tensor: shape=(2,), dtype=float32, numpy=array([1.       , 0.9999999],␣
↪dtype=float32)>)
```

7.4.2 モデル

これで、ソフトマックス回帰モデルを実装するために必要なものはすべて揃った。線形回帰の例と同様に、各データ例は固定長ベクトルで表現される。ここでの生データは 28×28 ピクセルの画像からなるので、各画像を平坦化し、長さ

784 のベクトルとして扱いる。後の章では、空間構造をより自然に活用できる畳み込みニューラルネットワークを紹介する。

ソフトマックス回帰では、ネットワークの出力数はクラス数と等しくなければならない。データセットには 10 クラスあるので、ネットワークの出力次元は 10 である。したがって、重みは 784×10 の行列と、バイアス用の 1×10 の行ベクトルから構成される。線形回帰と同様に、重み `W` はガウスノイズで初期化する。バイアスはゼロで初期化する。

```
class SoftmaxRegressionScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = torch.normal(0, sigma, size=(num_inputs, num_outputs),
                              requires_grad=True)
        self.b = torch.zeros(num_outputs, requires_grad=True)

    def parameters(self):
        return [self.W, self.b]
```

```
class SoftmaxRegressionScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = np.random.normal(0, sigma, (num_inputs, num_outputs))
        self.b = np.zeros(num_outputs)
        self.W.attach_grad()
        self.b.attach_grad()

    def collect_params(self):
        return [self.W, self.b]
```

```
class SoftmaxRegressionScratch(d2l.Classifier):
    num_inputs: int
    num_outputs: int
    lr: float
    sigma: float = 0.01

    def setup(self):
        self.W = self.param('W', nn.initializers.normal(self.sigma),
                            (self.num_inputs, self.num_outputs))
        self.b = self.param('b', nn.initializers.zeros, self.num_outputs)
```

```
class SoftmaxRegressionScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = tf.random.normal((num_inputs, num_outputs), 0, sigma)
        self.b = tf.zeros(num_outputs)
        self.W = tf.Variable(self.W)
        self.b = tf.Variable(self.b)
```

以下のコードでは、ネットワークが各入力をどのように出力へ写像するかを定義する。バッチ内の各 28×28 ピクセル画像は、モデルに通す前に `reshape` を使ってベクトルに平坦化していることに注意されたい。

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = d2l.reshape(X, (-1, self.W.shape[0]))
    return softmax(d2l.matmul(X, self.W) + self.b)
```

7.4.3 交差エントロピー損失

次に、交差エントロピー損失関数を実装する必要がある (Section 7.1.2 で導入した)。これは深層学習全体の中でも最も一般的な損失関数かもしれない。現時点では、深層学習の応用のうち、回帰問題よりも分類問題として自然に定式化できるものの方がはるかに多くある。

交差エントロピーは、真のラベルに割り当てられた予測確率の負の対数尤度を取ることを思い出そう。効率のため、Python の for ループは避け、代わりにインデックス参照を使う。特に、$\mathbf{y}$ の one-hot エンコーディングにより、$\hat{\mathbf{y}}$ の対応する項を選択できる。

これを確認するために、3 クラスに対する予測確率を持つ 2 つのデータ例 `y_hat` と、それに対応するラベル `y` を作成する。正しいラベルはそれぞれ 0 と 2（すなわち第 1 クラスと第 3 クラス）である。`y` を `y_hat` 内の確率のインデックスとして使うことで、項を効率よく取り出せる。

```
y = d2l.tensor([0, 2])
y_hat = d2l.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
tensor([0.1000, 0.5000])
```

```
y = d2l.tensor([0, 2])
y_hat = d2l.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
array([0.1, 0.5])
```

```
y = d2l.tensor([0, 2])
y_hat = d2l.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
Array([0.1, 0.5], dtype=float32)
```

```
y_hat = tf.constant([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y = tf.constant([0, 2])
tf.boolean_mask(y_hat, tf.one_hot(y, depth=y_hat.shape[-1]))
```

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.1, 0.5], dtype=float32)>
```

これで、選択した確率の対数を平均することで、交差エントロピー損失関数を実装できる。

```
def cross_entropy(y_hat, y):
    return -d2l.reduce_mean(d2l.log(y_hat[list(range(len(y_hat))), y]))

cross_entropy(y_hat, y)
```

```
tensor(1.4979)
```

```
def cross_entropy(y_hat, y):
    return -d2l.reduce_mean(d2l.log(y_hat[list(range(len(y_hat))), y]))

cross_entropy(y_hat, y)
```

```
array(1.4978662)
```

```
def cross_entropy(y_hat, y):
    return -d2l.reduce_mean(d2l.log(y_hat[list(range(len(y_hat))), y]))

cross_entropy(y_hat, y)
```

```
Array(1.4978662, dtype=float32)
```

```
def cross_entropy(y_hat, y):
    return -tf.reduce_mean(tf.math.log(tf.boolean_mask(
        y_hat, tf.one_hot(y, depth=y_hat.shape[-1]))))

cross_entropy(y_hat, y)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=1.4978662>
```

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

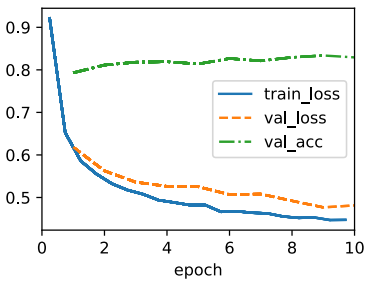
```
@d2l.add_to_class(SoftmaxRegressionScratch)
@partial(jax.jit, static_argnums=(0))
def loss(self, params, X, y, state):
    def cross_entropy(y_hat, y):
        return -d2l.reduce_mean(d2l.log(y_hat[list(range(len(y_hat))), y]))
    y_hat = state.apply_fn({'params': params}, *X)
    # 返される空の辞書は補助データのプレースホルダであり、
    # 後で（たとえば batch norm のために）使われます
    return cross_entropy(y_hat, y), {}
```

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

7.4.4 学習

Section 6.4 で定義した `fit` メソッドを再利用して、10 エポックでモデルを学習する。エポック数（`max_epochs`）、ミニバッチサイズ（`batch_size`）、学習率（`lr`）は調整可能なハイパーパラメータであることに注意されたい。つまり、これらの値は主たる学習ループの中で学習されるわけではないが、学習性能と汎化性能の両方において、モデルの性能に影響を与える。実際には、データの 検証分割に基づいてこれらの値を選び、最終的には テスト分割で最終モデルを評価したいだろう。Section 6.6.3 で述べたように、Fashion-MNIST のテストデータを検証セットとして扱い、この分割上で検証損失と検証精度を報告する。

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



7.4.5 予測

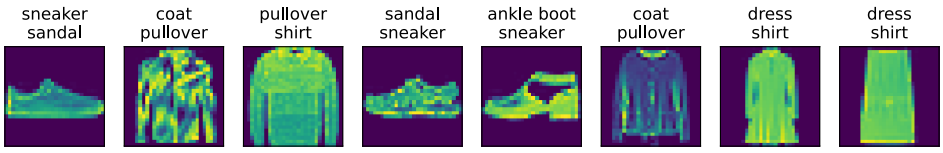
学習が完了したので、モデルはいくつかの画像を分類する準備ができた。

```
X, y = next(iter(data.val_dataloader()))
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    preds = d2l.argmax(model(X), axis=1)
if tab.selected('jax'):
    preds = d2l.argmax(model.apply({'params': trainer.state.params}, X),␣
↪axis=1)
preds.shape
```

```
torch.Size([256])
```

私たちがより関心を持つのは、誤ってラベル付けされた画像である。実際のラベル（テキスト出力の 1 行目）とモデルの予測（テキスト出力の 2 行目）を比較することで、それらを可視化する。

```
wrong = d2l.astype(preds, y.dtype) != y
X, y, preds = X[wrong], y[wrong], preds[wrong]
labels = [a+'\n'+b for a, b in zip(
    data.text_labels(y), data.text_labels(preds))]
data.visualize([X, y], labels=labels)
```



7.4.6 まとめ

ここまでで、線形回帰と分類問題を解く経験を少し積んできた。これにより、統計モデリングの 1960〜1970 年代におけるいわば最先端に到達したと言ってよいだろう。次の節では、このモデルを深層学習フレームワークを活用してはるかに効率よく実装する方法を示す。

7.4.7 演習

1. この節では、ソフトマックス演算の数学的定義に基づいてソフトマックス関数を直接実装した。Section 7.1 で述べたように、これは数値的不安定性を引き起こす可能性がある。
    1. 入力に値 100 が含まれていても `softmax` が正しく動作するか確認せよ。
    2. すべての入力の最大値が −100 より小さい場合でも `softmax` が正しく動作するか確認せよ。
    3. 引数の最大要素に対する相対値を見ることで修正を実装せよ。

2. 交差エントロピー損失関数 $\sum_i y_i \log \hat{y}_i$ の定義に従う `cross_entropy` 関数を実装せよ。
   1. この節のコード例で試してみよ。
   2. なぜより遅く動作すると考えられるか。
   3. それを使うべきか。どのような場合に使うのが理にかなっているか。
   4. 何に注意する必要があるか。ヒント: 対数の定義域を考えよ。
3. 最もありそうなラベルを返すことは、常に良い考えだろうか。たとえば、医療診断でこれを行うだろうか。どのように対処しようとするか。
4. いくつかの特徴量に基づいて次の単語を予測するためにソフトマックス回帰を使いたいと仮定する。大きな語彙から生じる問題にはどのようなものがあるか。
5. この節のコードのハイパーパラメータをいろいろ試してみよ。特に:
   1. 学習率を変えたときに検証損失がどのように変化するかをプロットせよ。
   2. ミニバッチサイズを変えると検証損失と学習損失は変化するか。効果が見えるまでに、どれくらい大きくまたは小さくする必要があるか。

7.5 Softmax 回帰の簡潔な実装

高水準の深層学習フレームワークが線形回帰の実装を容易にしたのと同様に (Section 6.5 を参照)、ここでも同様に便利である。

```
from d2l import torch as d2l
import torch
from torch import nn
from torch.nn import functional as F
```

```
from d2l import mxnet as d2l
from mxnet import gluon, init, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from functools import partial
import jax
from jax import numpy as jnp
import optax
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

7.5.1 モデルの定義

Section 6.5 と同様に、組み込み層を使って全結合層を構成する。その後、組み込みの `__call__` メソッドが、ネットワークを入力に適用する必要があるたびに `forward` を呼び出す。

`Flatten` 層を使って、4 階テンソル `X` を 2 階テンソルに変換する。第 1 軸の次元は変えない。

```
class SoftmaxRegression(d2l.Classifier):  #@save
    """The softmax regression model."""
    def __init__(self, num_outputs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(),
                                 nn.LazyLinear(num_outputs))

    def forward(self, X):
        return self.net(X)
```

```
class SoftmaxRegression(d2l.Classifier):  #@save
    """The softmax regression model."""
    def __init__(self, num_outputs, lr):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Dense(num_outputs)
            self.net.initialize()
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential()
            self.net.add(tf.keras.layers.Flatten())
            self.net.add(tf.keras.layers.Dense(num_outputs))

    def forward(self, X):
        return self.net(X)
```

```
class SoftmaxRegression(d2l.Classifier):  #@save
    num_outputs: int
    lr: float

    @nn.compact
    def __call__(self, X):
        X = X.reshape((X.shape[0], -1))  # Flatten
```

(continues on next page)

(continued from previous page)

```
        X = nn.Dense(self.num_outputs)(X)
        return X
```

```
class SoftmaxRegression(d2l.Classifier):  #@save
    """The softmax regression model."""
    def __init__(self, num_outputs, lr):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Dense(num_outputs)
            self.net.initialize()
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential()
            self.net.add(tf.keras.layers.Flatten())
            self.net.add(tf.keras.layers.Dense(num_outputs))

    def forward(self, X):
        return self.net(X)
```

7.5.2 Softmax の再考

Section 7.4 では、モデルの出力を計算し、クロスエントロピー損失を適用した。数学的にはこれはまったく妥当であるが、指数計算における数値的なアンダーフローとオーバーフローのため、計算上は危険である。

softmax 関数は $\hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$ によって確率を計算することを思い出そう。もし o_k のいくつかが非常に大きい、つまり絶対値の大きな正の値であれば、$\exp(o_k)$ は特定のデータ型で表現できる最大値よりも大きくなるかもしれない。これをオーバーフローと呼ぶ。同様に、引数がすべて非常に大きな負の値であれば、アンダーフローが起こりる。たとえば、単精度浮動小数点数はおおよそ 10^{-38} から 10^{38} の範囲をカバーする。したがって、$\mathbf{o}$ の最大項が $[-90, 90]$ の範囲外にあると、結果は安定しない。この問題を回避する方法は、すべての要素から $\bar{o} \stackrel{\text{def}}{=} \max_k o_k$ を引くことである。

$$\hat{y}_j = \frac{\exp o_j}{\sum_k \exp o_k} = \frac{\exp(o_j - \bar{o}) \exp \bar{o}}{\sum_k \exp(o_k - \bar{o}) \exp \bar{o}} = \frac{\exp(o_j - \bar{o})}{\sum_k \exp(o_k - \bar{o})}. \tag{7.5.1}$$

構成上、すべての j について $o_j - \bar{o} \leq 0$ であることがわかる。したがって、q クラス分類問題では、分母は区間 $[1, q]$ に収まりる。さらに、分子は 1 を超えないため、数値オーバーフローを防げる。数値アンダーフローは $\exp(o_j - \bar{o})$ が数値的に 0 と評価されるときにのみ起こりる。それでも、少し先で $\log \hat{y}_j$ を $\log 0$ として計算しようとすると問題が生じるかもしれない。特に、逆伝播では、忌まわしい NaN（Not a Number）の結果が画面いっぱいに現れる事態に直面するかもしれない。

幸いなことに、指数関数を計算しているにもかかわらず、最終的にはその対数を取る（クロスエントロピー損失を計算するとき）ことを意図しているため、救われる。softmax とクロスエントロピーを組み合わせることで、数値安定性の

問題を完全に回避できる。次が成り立ちる。

$$\log \hat{y}_j = \log \frac{\exp(o_j - \bar{o})}{\sum_k \exp(o_k - \bar{o})} = o_j - \bar{o} - \log \sum_k \exp(o_k - \bar{o}). \tag{7.5.2}$$

これにより、オーバーフローとアンダーフローの両方を避けられる。モデルの出力確率を評価したい場合に備えて、従来の softmax 関数も手元に置いておきたいところである。しかし、新しい損失関数に softmax 確率を渡す代わりに、単にロジットを渡し、クロスエントロピー損失関数の内部で softmax とその対数を一度に計算することで、“LogSumExp trick”[71] のような賢い処理を行える。

71 

```
@d2l.add_to_class(d2l.Classifier)  #@save
def loss(self, Y_hat, Y, averaged=True):
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    Y = d2l.reshape(Y, (-1,))
    if tab.selected('mxnet'):
        fn = gluon.loss.SoftmaxCrossEntropyLoss()
        l = fn(Y_hat, Y)
        return l.mean() if averaged else l
    if tab.selected('pytorch'):
        return F.cross_entropy(
            Y_hat, Y, reduction='mean' if averaged else 'none')
    if tab.selected('tensorflow'):
        fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
        return fn(Y, Y_hat)
```

```
@d2l.add_to_class(d2l.Classifier)  #@save
def loss(self, Y_hat, Y, averaged=True):
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    Y = d2l.reshape(Y, (-1,))
    if tab.selected('mxnet'):
        fn = gluon.loss.SoftmaxCrossEntropyLoss()
        l = fn(Y_hat, Y)
        return l.mean() if averaged else l
    if tab.selected('pytorch'):
        return F.cross_entropy(
            Y_hat, Y, reduction='mean' if averaged else 'none')
    if tab.selected('tensorflow'):
        fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
        return fn(Y, Y_hat)
```

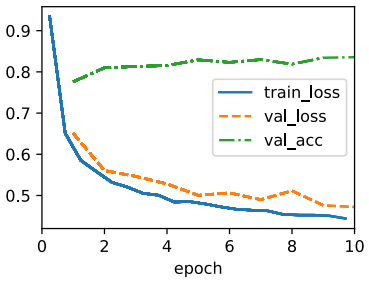
```
@d2l.add_to_class(d2l.Classifier)  #@save
@partial(jax.jit, static_argnums=(0, 5))
def loss(self, params, X, Y, state, averaged=True):
    # To be used later (e.g., for batch norm)
    Y_hat = state.apply_fn({'params': params}, *X,
                           mutable=False, rngs=None)
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    Y = d2l.reshape(Y, (-1,))
    fn = optax.softmax_cross_entropy_with_integer_labels
    # The returned empty dictionary is a placeholder for auxiliary data,
    # which will be used later (e.g., for batch norm)
    return (fn(Y_hat, Y).mean(), {}) if averaged else (fn(Y_hat, Y), {})
```

```
@d2l.add_to_class(d2l.Classifier)  #@save
def loss(self, Y_hat, Y, averaged=True):
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    Y = d2l.reshape(Y, (-1,))
    if tab.selected('mxnet'):
        fn = gluon.loss.SoftmaxCrossEntropyLoss()
        l = fn(Y_hat, Y)
        return l.mean() if averaged else l
    if tab.selected('pytorch'):
        return F.cross_entropy(
            Y_hat, Y, reduction='mean' if averaged else 'none')
    if tab.selected('tensorflow'):
        fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
        return fn(Y, Y_hat)
```

7.5.3 学習

次にモデルを学習する。Fashion-MNIST 画像を、784 次元の特徴ベクトルに平坦化して用いる。

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



これまでと同様に、このアルゴリズムはかなり正確な解に収束する。今回は以前よりも少ないコード行数でそれが実現できる。

7.5.4 まとめ

高水準 API は、数値安定性のような潜在的に危険な側面を利用者からうまく隠してくれるので非常に便利である。さらに、ごく少ないコード行数で簡潔にモデルを設計できるようにしてくれる。これは祝福であると同時に呪いでもある。明らかな利点は、統計の授業を一度も受けたことのないエンジニアでさえも非常に利用しやすくなることです（実際、彼らはこの本の想定読者の一部です）。しかし、鋭い部分を隠すことには代償もある。自分で新しく異なる構成要素を追加しようという意欲が削がれやすいのである。というのも、それを行うための身体的な記憶がほとんど身につかないからである。さらに、フレーム

ワークの保護用の緩衝材がすべての例外ケースを完全には覆いきれないときに、それを修正することも難しくなる。これもやはり、慣れの不足によるものである。

そのため、以下に続く多くの実装については、素朴な版と洗練された版の両方を確認することを強く勧める。理解しやすさを重視しているが、それでも実装は通常かなり高性能です（畳み込みはここでの大きな例外です）。私たちの意図は、あなたがフレームワークでは得られない新しいものを発明したときに、それを土台として発展させられるようにすることである。

7.5.5 演習

1. 深層学習では、FP64 倍精度（非常にまれに使われる）、FP32 単精度、BFLOAT16（圧縮表現に適している）、FP16（非常に不安定）、TF32（NVIDIA の新しい形式）、INT8 など、多くの異なる数値形式が使われる。数値アンダーフローやオーバーフローを起こさない指数関数の引数の最小値と最大値を求めなさい。

2. INT8 は、1 から 255 までの非ゼロ数からなる非常に制約の厳しい形式である。より多くのビットを使わずに、その動的範囲をどのように拡張できるか？通常の乗算と加算はそのまま使えますか？

3. 学習のエポック数を増やしなさい。しばらくすると検証精度が下がるのはなぜだろうか？それをどう修正できるか？

4. 学習率を増やすと何が起こりますか？いくつかの学習率について損失曲線を比較しなさい。どれがよりうまく機能するか？それはいつですか？

7.6 分類における汎化

これまで、複数出力と softmax 関数をもつ（線形）ニューラルネットワークを訓練することで、多クラス分類問題にどう取り組むかに焦点を当ててきた。モデルの出力を確率的予測として解釈し、モデルが（固定されたパラメータ集合に対して）実際のラベルに割り当てる負の対数尤度を計算する交差エントロピー損失関数を動機づけ、導出した。そして最後に、これらの道具を実際に用いて、訓練集合に対してモデルを当てはめた。しかし、いつものように、私たちの目標は、これまで見たことのないデータ（テスト集合）で経験的に評価される 一般的なパターンを学習することである。訓練集合で高い精度を達成しても、それ自体には何の意味もない。各入力が一意である場合（そして実際、高次元データセットの大半ではこれは真です）には、最初の訓練エポックでデータセットを丸暗記し、その後は新しい画像を見るたびにラベルを引き当てるだけで、訓練集合に対して完全な精度を達成できる。しかし、訓練例それぞれに対応する正確なラベルをそのまま記憶しても、新しい例をどう分類すべきかは

分からない。さらなる指針がなければ、新しい例に遭遇するたびに、私たちはランダムに推測するしかないかもしれない。

差し迫ったいくつかの疑問に、すぐに答える必要がある。

1. 母集団全体における分類器の精度をよく推定するには、どれくらいのテスト例が必要だろうか？
2. 同じテスト集合でモデルを繰り返し評価し続けると、何が起こるだろうか？
3. なぜ、線形モデルを訓練集合に当てはめることが、素朴な丸暗記方式よりもうまくいくと期待できるのだろうか？

Section 6.6 では線形回帰の文脈で過学習と汎化の基礎を紹介したが、本章ではもう少し踏み込み、統計的学習理論の基礎的な考え方をいくつか導入する。実は、汎化を *a priori* に保証できることがしばしばある。多くのモデルについて、そして汎化ギャップ ϵ に対する任意の上界を指定すると、訓練集合が少なくとも n 個のサンプルを含むなら経験誤差が真の誤差から ϵ 以内に収まるような、必要サンプル数 n をしばしば決定できる。しかもそれは、どのようなデータ生成分布に対しても成り立ちる。残念ながら、こうした保証は知的な構成要素としては非常に重要である一方で、深層学習の実践者にとっては実用上の価値が限られていることも分かっている。要するに、こうした保証は、深いニューラルネットワークの汎化を *a priori* に保証するには、ばかげたほど大量のデータ例（おそらく兆単位、あるいはそれ以上）が必要であることを示唆する。一方で、私たちが関心を持つタスクでは、深いニューラルネットワークがはるかに少ないデータ例（数千程度）で驚くほどよく汎化することも分かっている。そのため深層学習の実践者は、*a priori* の保証を完全に諦めることが多く、代わりに過去の類似問題でよく汎化した方法を用い、経験的評価を通じて *post hoc* に汎化を検証する。Section 8 では、汎化を再び取り上げ、深いニューラルネットワークが実際にうまく汎化する理由を説明しようとする膨大な科学文献への簡単な導入を行う。

7.6.1 テスト集合

すでにテスト集合を、汎化誤差を評価するためのゴールドスタンダードとして使い始めているので、まずはそのような誤差推定の性質について議論しよう。ここでは、どのようにして得られたかは気にせず、固定された分類器 f に注目する。さらに、分類器 f の訓練には使われていない、新しいデータ例からなるデータセット $\mathcal{D} = (\mathbf{x}^{(i)}, y^{(i)})_{i=1}^{n}$ を持っていると仮定する。$\mathcal{D}$ 上での分類器 f の経験誤差とは、予測 $f(\mathbf{x}^{(i)})$ が真のラベル $y^{(i)}$ と一致しない事例の割合にすぎず、次の式で与えられる。

$$\epsilon_{\mathcal{D}}(f) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}(f(\mathbf{x}^{(i)}) \neq y^{(i)}). \tag{7.6.1}$$

これに対して、母集団誤差とは、母集団全体における例の 期待割合です（確率密度関数 $p(\mathbf{x}, y)$ で特徴づけられるある分布 $P(X, Y)$ において）、そのデータ例

に対して分類器が真のラベルと異なる予測をする割合の期待値である。

$$\epsilon(f) = E_{(\mathbf{x},y)\sim P}\mathbf{1}(f(\mathbf{x}) \neq y) = \int\int \mathbf{1}(f(\mathbf{x}) \neq y)p(\mathbf{x}, y)\, d\mathbf{x}dy. \tag{7.6.2}$$

$\epsilon(f)$ は私たちが実際に知りたい量であるが、直接観測することはできない。大きな母集団の平均身長を、全員を測定しない限り直接観測できないのと同じである。私たちにできるのは、標本に基づいてこの量を推定することだけである。テスト集合 $\mathcal{D}$ が母集団全体を統計的に代表しているため、$\epsilon_{\mathcal{D}}(f)$ を母集団誤差 $\epsilon(f)$ の統計的推定量とみなせる。さらに、私たちが関心を持つ量 $\epsilon(f)$ は（確率変数 $\mathbf{1}(f(X) \neq Y)$ の）期待値であり、対応する推定量 $\epsilon_{\mathcal{D}}(f)$ は標本平均なので、母集団誤差の推定は、Section 5.6 で見た古典的な平均推定の問題にすぎない。

確率論における重要な古典的結果である 中心極限定理は、平均 μ と標準偏差 σ をもつ任意の分布から引かれた n 個のランダムサンプル $a_1, ..., a_n$ があるとき、サンプル平均 $\hat{\mu}$ は、サンプル数 n が無限大に近づくにつれて、真の平均を中心とし標準偏差が $\sigma/\sqrt{n}$ の正規分布に近づくことを保証する。これだけでも重要なことが分かりる。データ例の数が大きくなるにつれて、テスト誤差 $\epsilon_{\mathcal{D}}(f)$ は真の誤差 $\epsilon(f)$ に $O(1/\sqrt{n})$ の速度で近づくはずである。したがって、テスト誤差を 2 倍正確に推定したければ、4 倍大きいテスト集合を集めなければならない。テスト誤差を 100 分の 1 に減らしたければ、1 万倍大きいテスト集合を集める必要がある。一般に、このような $O(1/\sqrt{n})$ の速度は、統計で期待できる最良のものである。

テスト誤差 $\epsilon_{\mathcal{D}}(f)$ が真の誤差 $\epsilon(f)$ に収束する漸近的な速度について分かったので、いくつかの重要な詳細に目を向けよう。関心のある確率変数 $\mathbf{1}(f(X) \neq Y)$ は 0 と 1 の値しか取れないので、ベルヌーイ確率変数である。その特徴は、1 を取る確率を表すパラメータによって決まりる。ここで 1 は分類器が誤りを犯したことを意味するので、この確率変数のパラメータは実際には真の誤り率 $\epsilon(f)$ である。ベルヌーイ分布の分散 σ^2 は、そのパラメータ（ここでは $\epsilon(f)$）に応じて $\epsilon(f)(1 - \epsilon(f))$ という式で与えられる。$\epsilon(f)$ は最初は未知であるが、1 を超えることはない。この関数を少し調べると、分散は真の誤り率が 0.5 に近いときに最大となり、0 や 1 に近いときにはずっと小さくなりうることが分かりる。これは、誤差 $\epsilon(f)$ の推定値 $\epsilon_{\mathcal{D}}(f)$ の漸近標準偏差が（n 個のテストサンプルの選び方に関して）$\sqrt{0.25/n}$ を超えないことを意味する。

この速度が、有限個のサンプルしか持たない場合ではなく、テスト集合サイズが無限大に近づくときの振る舞いを特徴づけているという事実を無視すれば、テスト誤差 $\epsilon_{\mathcal{D}}(f)$ が母集団誤差 $\epsilon(f)$ を近似し、その 1 標準偏差が ± 0.01 の区間に対応するようにしたければ、およそ 2500 サンプルを集めるべきだと分かりる。その範囲に 2 標準偏差を収め、したがって $\epsilon_{\mathcal{D}}(f) \in \epsilon(f) \pm 0.01$ であると 95% の確信を持ちたければ、1 万サンプルが必要になる！

これは、多くの機械学習ベンチマークにおけるテスト集合のサイズに相当する。毎年、何千もの応用深層学習論文が、0.01 以下の誤差率改善を大きく取り上げて発表されていると知ると、驚くかもしれない。もちろん、誤差率が 0 にずっと近い場合には、0.01 の改善は確かに大きな意味を持ちえる。

ここまでの分析の厄介な点の一つは、実際には漸近的なこと、すなわち $\epsilon_{\mathcal{D}}$ と ϵ の関係がサンプルサイズの増加に伴ってどう変化するかしか教えてくれないことである。幸いなことに、確率変数が有界であるため、Hoeffding（1963）による不等式を適用することで、有効な有限サンプル境界を得ることができる。

$$P(\epsilon_{\mathcal{D}}(f) - \epsilon(f) \geq t) < \exp\left(-2nt^2\right). \tag{7.6.3}$$

推定値 $\epsilon_{\mathcal{D}}(f)$ と真の誤差率 $\epsilon(f)$ の距離 t が 0.01 を超えないことを 95% の信頼度で結論づけるために必要な最小データセットサイズを解くと、上の漸近解析が示唆した 10,000 例に対して、およそ 15,000 例が必要であることが分かりる。統計をさらに深く学べば、この傾向は一般に成り立つことが分かるだろう。有限サンプルでも成り立つ保証は、通常、やや保守的である。とはいえ、全体として見ると、これらの数値はそれほど離れていない。これは、厳密な法廷で通用する保証ではなくても、おおよその見積もりを与えるという漸近解析の一般的な有用性を反映している。

7.6.2 テスト集合の再利用

ある意味では、これで経験的機械学習研究を進める準備は整った。実用上のほとんどすべてのモデルは、テスト集合の性能に基づいて開発・検証されており、あなたは今やテスト集合の達人である。任意の固定された分類器 f について、そのテスト誤差 $\epsilon_{\mathcal{D}}(f)$ を評価する方法を知っており、その母集団誤差 $\epsilon(f)$ について何が言えて、何が言えないかを正確に理解している。

では、この知識を使って最初のモデル f_1 を訓練しようとしているとしよう。分類器の誤り率の性能について、どれだけの確信が必要かを知っているので、上の解析を用いてテスト集合として確保すべき適切なデータ例の数を決める。さらに、Section 6.6 の教訓を真剣に受け止め、すべての予備解析、ハイパーパラメータ調整、さらには複数の競合するモデルアーキテクチャの選択までも検証集合上で行うことで、テスト集合の神聖さを守ったとしよう。最後に、モデル f_1 をテスト集合で評価し、対応する信頼区間付きで母集団誤差の偏りのない推定値を報告する。

ここまではすべて順調に見える。しかしその夜、あなたは午前 3 時に目を覚まし、新しいモデリング手法についての素晴らしいアイデアを思いつきる。翌日、その新しいモデルを実装し、検証集合でハイパーパラメータを調整すると、新しいモデル f_2 は動くだけでなく、その誤り率は f_1 よりもかなり低いように見える。しかし、発見の興奮は、最終評価の準備をするにつれて突然薄れていく。テスト集合がないのです！

元のテスト集合 $\mathcal{D}$ はまだサーバ上にあるが、今や 2 つの大きな問題に直面している。第一に、テスト集合を収集したとき、単一の分類器 f を評価するという仮定のもとで必要な精度を決めていた。しかし、同じテスト集合で複数の分類器 $f_1, ..., f_k$ を評価することにすると、偽発見の問題を考えなければならない。以前は、単一の分類器 f について $\epsilon_{\mathcal{D}}(f) \in \epsilon(f) \pm 0.01$ であると 95% の確率で言えるかもしれず、その場合、誤解を招く結果が出る確率はわずか 5% でした。し

かし k 個の分類器が混在すると、そのうち少なくとも 1 つについてテスト集合上の性能が誤解を招くものでないことを保証するのは難しくなる。20 個の分類器を検討していると、少なくとも 1 つが誤解を招くスコアを受け取っていない可能性を排除する力が、まったくないかもしれない。この問題は多重仮説検定に関係しており、統計学には膨大な文献があるにもかかわらず、科学研究を悩ませ続ける根深い問題である。

それでも十分に心配でないなら、その後の評価で得られる結果を疑うべき特別な理由がある。テスト集合の性能に関する私たちの解析は、分類器がテスト集合に一切触れずに選ばれたという仮定に基づいており、そのためテスト集合を母集団からランダムに抽出されたものとみなせた。ここでは、複数の関数をテストしているだけでなく、後続の関数 f_2 は f_1 のテスト集合性能を観測した後に選ばれている。一度テスト集合からの情報がモデル作成者に漏れたら、厳密な意味では、もはや真のテスト集合ではありえない。この問題は *adaptive overfitting* と呼ばれ、近年、学習理論家や統計学者の間で強い関心を集める話題として浮上している (Dwork *et al.*, 2015)。幸いなことに、ホールドアウト集合からすべての情報を漏らしてしまうことは可能であり、理論上の最悪ケースは暗いものであるが、こうした解析は保守的すぎるのかもしれない。実践では、本物のテスト集合を作成し、それを参照する頻度をできるだけ少なくし、信頼区間を報告するときには多重仮説検定を考慮し、賭け金が高くデータセットサイズが小さいときには、より強く警戒するのがよいだろう。一連のベンチマーク課題を実行するときには、各ラウンドの後に古いテスト集合を検証集合へ格下げできるよう、複数のテスト集合を維持するのがよい実践である。

7.6.3 統計的学習理論

簡単に言えば、テスト集合こそが私たちに本当にあるものである。しかし、この事実は奇妙なほど満足感に欠ける。第一に、私たちはめったに 真のテスト集合を持たない。 もし私たち自身がデータセットを作成しているのでなければ、他の誰かがすでに、私たちが「テスト集合」だと思っているものに対して自分の分類器を評価している可能性が高いからである。そして、たとえ先に使える立場にあったとしても、その後のモデリング試行を、数値を信頼できないというじわじわした不安なしに評価したいと、すぐに不満を感じるようになる。さらに、真のテスト集合であっても、分類器が実際に母集団へ汎化したかどうかを *post hoc* に教えてくれるだけで、*a priori* に汎化すると期待すべき理由があるかどうかは教えてくれない。

こうした懸念を踏まえると、統計的学習理論の魅力が十分に見えてくるかもしれない。これは機械学習の数学的な下位分野であり、その研究者たちは、経験データで訓練されたモデルがなぜ／いつ未見データへ汎化できるのかを説明する根本原理を明らかにしようとしている。統計的学習研究の主要な目的の一つは、汎化ギャップを上界づけることであり、モデルクラスの性質とデータセット中のサンプル数との関係を明らかにすることである。

学習理論家は、学習された分類器 f_S の 経験誤差 $\epsilon_S(f_S)$ と、同じ分類器の母集団全体における真の誤差 $\epsilon(f_S)$ との差を上界づけようとする。ここで f_S は訓練集合 S 上で訓練され、かつ評価された分類器である。これは、先ほど扱った

評価問題に似ているが、大きな違いがある。前節では、分類器 f は固定されており、私たちに必要だったのは評価目的のデータセットだけでした。そして実際、任意の固定された分類器は汎化する。その誤差は、（これまで見たことのない）データセット上で母集団誤差の偏りのない推定値になる。しかし、分類器が同じデータセットで訓練され、評価されるとき、私たちは何を言えるのだろうか？訓練誤差がテスト誤差に近いと、いつ確信できるのだろうか？

学習された分類器 f_S が、あらかじめ指定された関数集合 $\mathcal{F}$ から選ばれなければならないとしよう。テスト集合についての議論を思い出すと、単一の分類器の誤差を推定するのは簡単であるが、分類器の集合を考え始めると話が厄介になることが分かりる。たとえ任意の 1 つの（固定された）分類器の経験誤差が高い確率で真の誤差に近いとしても、分類器の集合を考えると、そのうち たった *1* つがひどく誤って推定された誤差を受け取る可能性を心配しなければならない。懸念されるのは、そのような分類器を選んでしまい、母集団誤差を大きく過小評価してしまうことである。さらに、線形モデルでさえ、パラメータが連続値であるため、通常は無限個の関数からなるクラス（$|\mathcal{F}| = \infty$）から選んでいることになる。

この問題に対する野心的な解決策の一つは、一様収束を証明するための解析的道具を開発することである。すなわち、高い確率で、クラス内のすべての分類器 $f \in \mathcal{F}$ の経験誤差率が、その真の誤差率へ 同時に収束することを示すことである。言い換えれば、少なくとも $1-\delta$ の確率で（ある小さな δ に対して）、クラス $\mathcal{F}$ に属するすべての分類器の誤差率 $\epsilon(f)$ が、ある小さな量 α よりも大きく誤推定されない、と述べられる理論原理を求めているのである。明らかに、こうした主張をすべてのモデルクラス $\mathcal{F}$ に対して行うことはできない。常に経験誤差 0 を達成する一方で、母集団全体ではランダム推測を上回ることのない、丸暗記マシンのクラスを思い出そう。

ある意味で、丸暗記器のクラスは柔軟すぎるのである。そのような一様収束結果は、到底成り立ちえない。一方で、固定された分類器は役に立たない。 それは完全に汎化するが、訓練データにもテストデータにも適合しないからである。したがって学習の中心問題は歴史的に、訓練データにはよく適合するが過学習の危険がある、より柔軟な（分散の高い）モデルクラスと、うまく汎化するが過小適合の危険がある、より硬直的な（バイアスの高い）モデルクラスとのトレードオフとして定式化されてきた。学習理論における中心的な課題は、モデルがこのスペクトルのどこに位置するかを定量化し、それに対応する保証を与える適切な数学的解析を発展させることでした。

一連の画期的な論文で、Vapnik と Chervonenkis は、相対頻度の収束に関する理論を、より一般的な関数クラスへ拡張した (Vapnik and Chervonenkis, 1964, Vapnik and Chervonenkis, 1968, Vapnik and Chervonenkis, 1971, Vapnik and Chervonenkis, 1981, Vapnik and Chervonenkis, 1991, Vapnik and Chervonenkis, 1974)。この流れの重要な貢献の一つが Vapnik–Chervonenkis（VC）次元であり、これはモデルクラスの複雑さ（柔軟さ）の一つの尺度を表す。さらに、彼らの主要な結果の一つは、経験誤差と母集団誤差の差を、VC 次元とサンプル数の関数として上界づけ

る。

$$P\left(R[p,f] - R_{\text{emp}}[\mathbf{X},\mathbf{Y},f] < \alpha\right) \geq 1-\delta \text{ for } \alpha \geq c\sqrt{(\text{VC} - \log\delta)/n}. \tag{7.6.4}$$

ここで $\delta > 0$ はこの境界が破られる確率であり、α は汎化ギャップの上界、n はデータセットサイズである。最後に、$c > 0$ は、被りうる損失のスケールのみに依存する定数である。この境界の使い方の一つは、望ましい δ と α の値を代入して、どれだけのサンプルを集めるべきかを決めることである。VC 次元は、任意の（二値の）ラベル付けを割り当てることができ、かつその各ラベル付けに一致するモデル f がクラス内に存在するような、データ点の最大数を定量化する。たとえば、d 次元入力に対する線形モデルの VC 次元は $d+1$ である。2 次元で 3 点に対しては、直線が任意のラベル付けを実現できることは容易に分かりるが、4 点に対してはできない。残念ながら、この理論はより複雑なモデルに対しては過度に悲観的になりがちで、この保証を得るには、実際に望ましい誤差率を達成するのに必要な数よりもはるかに多くのデータ例が通常必要になる。また、モデルクラスと
delta を固定すると、誤差率はやはり通常の
mathcalO(1/
sqrtn) の速度で減少することに注意されたい。n の観点では、これ以上うまくやれるとは思えない。しかし、モデルクラスを変えると、VC 次元は汎化ギャップについて悲観的な見方を与えることがある。

7.6.4 まとめ

モデルを評価する最も直接的な方法は、これまで見たことのないデータからなるテスト集合を参照することである。テスト集合による評価は真の誤差の偏りのない推定値を与え、テスト集合が大きくなるにつれて望ましい $O(1/\sqrt{n})$ の速度で収束する。近似的な信頼区間は、厳密な漸近分布に基づいて、あるいは（より保守的な）有限サンプル保証に基づいて、有効な有限サンプル信頼区間として与えられる。実際、テスト集合による評価は現代機械学習研究の基盤である。しかし、テスト集合は真のテスト集合であることはまれです（複数の研究者によって何度も使われるため）。同じテスト集合が複数のモデルの評価に使われると、偽発見の制御が難しくなることがある。これは理論上、大きな問題を引き起こしえる。実践では、この問題の重大さは、対象となるホールドアウト集合のサイズや、それが単にハイパーパラメータ選択に使われるだけなのか、それともより直接的に情報が漏れているのかに依存する。それでも、本物のテスト集合（あるいは複数）を整備し、どれくらいの頻度で使うかについてはできるだけ保守的であるのがよい実践である。

より満足のいく解決策を提供しようとして、統計的学習理論家たちは、モデルクラス全体にわたる一様収束を保証する方法を開発してきた。もし本当に、すべてのモデルの経験誤差が同時に真の誤差へ収束するなら、訓練誤差を最小化して最もよく性能を示すモデルを自由に選んでも、それがホールドアウトデータ上でも同様によく性能を示すと分かっていることになる。重要なのは、そのような結果は必ずモデルクラスの何らかの性質に依存するということである。Vladimir Vapnik と Alexey Chernovenkis は VC 次元を導入し、VC クラスに属する

すべてのモデルに対して成り立つ一様収束結果を提示した。クラス内のすべてのモデルの訓練誤差は、（同時に）その真の誤差に近いことが保証され、さらに $O(1/\sqrt{n})$ の速度でより近づくことが保証される。VC 次元の革命的な発見に続いて、数多くの代替的な複雑さ尺度が提案され、それぞれが同様の汎化保証を可能にした。関数の複雑さを測るいくつかの高度な方法についての詳細な議論は Boucheron *et al.* (2005) を参照されたい。残念ながら、これらの複雑さ尺度は統計理論では広く有用な道具となった一方で、深いニューラルネットワークがなぜ汎化するのかを説明するためには、（そのまま適用すると）無力であることが分かっている。深いニューラルネットワークはしばしば数百万個以上のパラメータを持ち、大量の点にランダムなラベルを容易に割り当てることができる。それにもかかわらず、実用上の問題ではよく汎化し、しかも驚くべきことに、VC 次元が高くなるにもかかわらず、より大きくより深いほど、しばしばよりよく汎化する。次章では、深層学習の文脈で汎化を再び取り上げる。

7.6.5 演習

1. 固定されたモデル f の誤差を 0.0001 以内で、かつ 99.9% を超える確率で推定したいとき、何個のサンプルが必要だろうか？
2. ある人がラベル付きテスト集合 $\mathcal{D}$ を持っており、ラベルなしの入力（特徴量）だけを公開しているとする。いま、各ラベルなし入力に対してモデル f（モデルクラスに制限はない）を実行し、対応する誤差 $\epsilon_{\mathcal{D}}(f)$ を受け取ることでしか、テスト集合のラベルにアクセスできないとする。テスト集合全体を漏らしてしまい、その結果、真の誤差に関係なく誤差 0 であるかのように見せられるようになるまでに、何個のモデルを評価する必要があるだろうか？
3. 5 次多項式のクラスの VC 次元はいくつだろうか？
4. 2 次元データ上の軸平行長方形の VC 次元はいくつだろうか？

7.7 環境と分布シフト

前の節では、機械学習のさまざまな実践的応用を通して、多様なデータセットにモデルを当てはめてきた。しかし、そもそもデータがどこから来たのか、あるいはモデルの出力を最終的に何に使うのか、という点について立ち止まって考えることはなかった。データを手にした機械学習開発者は、こうした根本的な問題を考えずにモデル開発へと急ぎがちである。

失敗した機械学習の導入の多くは、この見落としに起因している。テストセット精度で見ると非常に優秀に見えるのに、本番導入するとデータ分布が突然変化して壊滅的に失敗することがある。さらに厄介なのは、モデルの導入そのものがデータ分布を乱す引き金になる場合があることである。たとえば、ある人がローンを返済するか延滞するかを予測するモデルを学習し、申請者の履物の

選択が延滞リスクと関連している（オックスフォード靴は返済、スニーカーは延滞）と分かったとする。すると私たちは、オックスフォード靴を履いている申請者にはローンを認め、スニーカーを履いている申請者はすべて拒否したくなるかもしれない。

この場合、パターン認識から意思決定への軽率な飛躍と、環境を批判的に考慮しない姿勢が、悲惨な結果を招く可能性がある。まず、履物に基づいて判断し始めた瞬間に、顧客はそのことに気づいて行動を変えるだろう。やがて、信用力が改善したわけでもないのに、すべての申請者がオックスフォード靴を履くようになるはずである。少し立ち止まって考えてみよ。機械学習の多くの応用で同様の問題が起こりうるからである。モデルに基づく意思決定を環境に持ち込むことで、モデルそのものを壊してしまうかもしれないのである。

この節だけでこれらの話題を完全に扱うことはできないが、ここではよくある懸念をいくつか示し、そのような状況を早期に検知し、被害を軽減し、機械学習を責任ある形で使うために必要な批判的思考を促したいと思う。解決策の中には単純なものもある（「正しい」データを求めること）、技術的に難しいものもある（強化学習システムを実装すること）、また、統計的予測の領域を超えて、アルゴリズムの倫理的な適用に関する難しい哲学的問題に取り組む必要があるものもある。

7.7.1 分布シフトの種類

まずは受動的な予測設定に立ち返り、データ分布がどのように変化しうるか、そしてモデル性能を救うために何ができるかを考える。古典的な設定の一つでは、訓練データがある分布 $p_S(\mathbf{x}, y)$ からサンプリングされた一方で、テストデータは別の分布 $p_T(\mathbf{x}, y)$ から得られるラベルなしのデータ例で構成されると仮定する。この時点で、厳しい現実に向き合わなければならない。p_S と p_T の関係について何の仮定も置かないなら、頑健な分類器を学習することは不可能である。

犬と猫を区別したい二値分類問題を考えよう。分布が任意の形で変化しうるなら、入力分布は一定のまま $p_S(\mathbf{x}) = p_T(\mathbf{x})$ である一方、ラベルだけがすべて反転するという病的なケースを許してしまいます：$p_S(y \mid \mathbf{x}) = 1 - p_T(y \mid \mathbf{x})$。言い換えると、神が突然、将来はすべての「猫」が犬であり、以前「犬」と呼んでいたものが猫であると決めたとしても　入力分布 $p(\mathbf{x})$ に何の変化もないまま　この状況を、分布がまったく変わらなかった場合と区別することはできない。

幸いなことに、将来データがどのように変化しうるかについていくつかの制約付き仮定を置けば、原理に基づくアルゴリズムがシフトを検知し、場合によってはその場で適応して、元の分類器の精度を上回ることさえできる。

共変量シフト

分布シフトのカテゴリの中でも、共変量シフトは最も広く研究されているものかもしれない。ここでは、入力分布は時間とともに変化しうるが、ラベル付け

関数、すなわち条件付き分布 $P(y \mid \mathbf{x})$ は変化しないと仮定する。統計学では、共変量（特徴量）の分布が変化することからこの問題を *covariate shift* と呼ぶ。因果性を持ち出さずに分布シフトを論じることもできるが、$\mathbf{x}$ が y を引き起こすと考える状況では、共変量シフトは自然な仮定である。

猫と犬を見分ける課題を考えてみよう。訓練データは Fig. 7.7.1 のような画像で構成されているかもしれない。

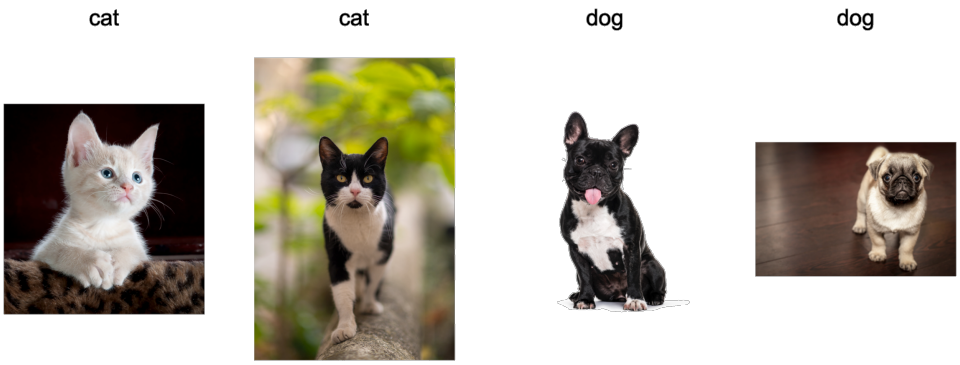


Fig. 7.7.1 猫と犬を区別するための訓練データ（イラスト: Lafeez Hossain / 500px / Getty Images; ilkermetinkursova / iStock / Getty Images Plus; GlobalP / iStock / Getty Images Plus; Musthafa Aboobakuru / 500px / Getty Images）。

テスト時には Fig. 7.7.2 の画像を分類するよう求められる。

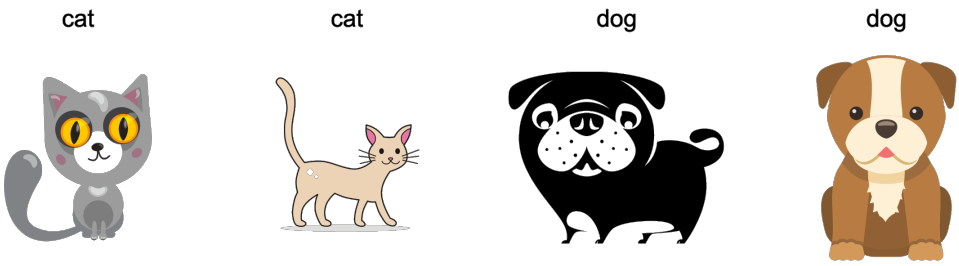


Fig. 7.7.2 猫と犬を区別するためのテストデータ（イラスト: SIBAS_minich / iStock / Getty Images Plus; Ghrzuzudu / iStock / Getty Images Plus; id-work / DigitalVision Vectors / Getty Images; Yime / iStock / Getty Images Plus）。

訓練セットは写真で構成されているのに対し、テストセットには漫画しか含まれていない。テストセットと大きく異なる特徴を持つデータセットで学習すると、新しいドメインにどう適応するかという一貫した計画がない限り、問題を引き起こす。

ラベルシフト

ラベルシフトはその逆の問題を表す。ここでは、ラベルの周辺分布 $P(y)$ は変化しうるが、クラス条件付き分布 $P(\mathbf{x} \mid y)$ はドメイン間で固定されていると仮定

する。ラベルシフトは、y が $\mathbf{x}$ を引き起こすと考えるときに妥当な仮定である。たとえば、診断名の相対的な有病率が時間とともに変化していても、症状（あるいは他の現れ方）から診断を予測したい場合がある。ここでは病気が症状を引き起こすので、ラベルシフトの仮定が適切である。退化的な場合には、ラベルシフトと共変量シフトの仮定が同時に成り立つこともある。たとえばラベルが決定論的であれば、たとえ y が $\mathbf{x}$ を引き起こす場合でも、共変量シフトの仮定は満たされる。興味深いことに、このような場合には、ラベルシフトの仮定から導かれる手法を使う方が有利なことが多いである。その理由は、これらの手法が入力のように見える高次元の対象ではなく、ラベルのように見える対象（しばしば低次元）を操作することが多いからである。深層学習では入力は画像のような高次元オブジェクトになりがちであるが、ラベルはカテゴリのようなより単純なオブジェクトであることが多いのである。

概念シフト

関連する問題として *concept shift* もある。これは、ラベルの定義そのものが変わりうるときに生じる。妙に聞こえるかもしれない　猫は 猫だろう？しかし、他のカテゴリでは時間とともに用法が変化する。精神疾患の診断基準、何が流行と見なされるか、職名などは、いずれもかなりの概念シフトの影響を受ける。実際、米国を地理的に移動してデータの収集元を変えると、Fig. 7.7.3 に示すように、炭酸飲料の呼び名の分布にかなりの概念シフトが見られる。

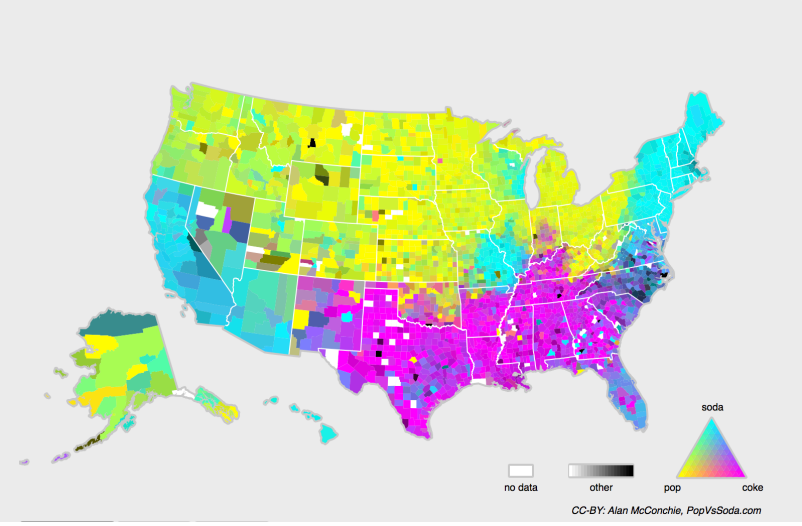


Fig. 7.7.3 米国における炭酸飲料名の概念シフト（CC-BY: Alan McConchie, PopVsSoda.com）。

機械翻訳システムを構築する場合、$P(y \mid \mathbf{x})$ の分布は場所によって異なるかもしれない。この問題は見つけるのが難しいことがある。時間的あるいは地理的な意味で、シフトが徐々にしか起こらないという知識を活用できると期待したくなる。

7.7.2 分布シフトの例

形式論やアルゴリズムに入る前に、共変量シフトや概念シフトが一見して明らかでない具体例をいくつか見てみよう。

医療診断

がんを検出するアルゴリズムを設計したいとしよう。健康な人と病気の人からデータを集めて、アルゴリズムを学習する。うまく動き、高い精度が得られたので、医療診断で成功するキャリアの準備ができたと結論づける。そう簡単ではない。

訓練データを生み出した分布と、実運用で遭遇する分布はかなり異なるかもしれない。これは、何年も前に著者の何人かが関わったある不運なスタートアップでも起こった。彼らは主に高齢男性に影響する病気の血液検査を開発しており、患者から採取した血液サンプルを使って研究したいと考えていた。しかし、すでにシステムに入っている病気の患者からよりも、健康な男性から血液サンプルを得る方がはるかに難しいのである。そこでそのスタートアップは、大学キャンパスの学生から血液提供を募り、検査開発における健康対照群として使った。その後、私たちにその病気を検出する分類器の構築を手伝ってもらえないかと依頼してきた。

私たちが説明したように、健康群と病気群をほぼ完全な精度で見分けること自体は確かに簡単である。しかし、それは被験者が年齢、ホルモンレベル、身体活動、食事、飲酒量、そして病気とは無関係な多くの要因で異なっていたからである。実際の患者ではそのようなことはまずない。このサンプリング手順のために、極端な共変量シフトが起こると予想された。さらに、このケースは従来の方法では修正できない可能性が高かったのである。要するに、彼らはかなりの金額を無駄にした。

自動運転車

ある会社が機械学習を活用して自動運転車を開発したいと考えたとする。ここで重要な構成要素の一つが路肩検出器である。実際の注釈付きデータは高価なので、彼らはゲームのレンダリングエンジンから得た合成データを追加の訓練データとして使うという（賢くもあり、疑わしくもある）アイデアを思いついた。これはレンダリングエンジンから得た「テストデータ」では非常によく機能した。ところが、実車では大失敗でした。調べてみると、路肩は非常に単純なテクスチャでレンダリングされていたのである。さらに重要なのは、路肩のすべてが同じテクスチャでレンダリングされており、路肩検出器はこの「特徴」をすぐに学習してしまったことである。

同様のことは、米軍が森林内の戦車検出を試みたときにも起こった。彼らは戦車のない森林の航空写真を撮り、その後戦車を森林に持ち込んで別の写真を撮った。分類器は完璧に動作しているように見えた。しかし実際には、木と影のある木を区別する方法を学んだだけだったのです　最初の写真は早朝に、2 回目の写真は正午に撮影されていた。

非定常分布

より微妙な状況は、分布がゆっくり変化する場合（非定常分布とも呼ばれます）で、モデルが十分に更新されないときに生じる。典型的な例をいくつか挙げる。

- 計算広告モデルを学習したのに、頻繁に更新しない（たとえば、iPad という新しい目立たないデバイスが発売されたことを取り込むのを忘れる）。
- スパムフィルタを作る。これまで見たすべてのスパムの検出にはうまくいく。しかし、スパマーが賢くなり、これまで見たことのないような新しいメッセージを作る。
- 商品推薦システムを作る。冬の間はうまく動くが、クリスマスが過ぎてもサンタ帽を推薦し続ける。

さらにいくつかの逸話

- 顔検出器を作る。すべてのベンチマークではうまくいく。ところがテストデータでは失敗する　問題の例は顔が画像全体を占めるクローズアップである（訓練セットにはそのようなデータがなかった）。
- 米国市場向けのウェブ検索エンジンを作り、それを英国で展開したい。
- 大規模なデータセットをまとめて画像分類器を学習する。クラス数が非常に多く、たとえば 1000 カテゴリがそれぞれ 1000 枚ずつで均等に表現されているデータセットを作る。その後、実世界にシステムを導入すると、写真の実際のラベル分布は明らかに一様ではない。

7.7.3 分布シフトの補正

すでに述べたように、訓練分布とテスト分布 $P(\mathbf{x}, y)$ が異なる場合は多くある。場合によっては、共変量シフト、ラベルシフト、概念シフトがあってもモデルがうまく動く幸運に恵まれる。別の場合には、シフトに対処するための原理に基づく戦略を用いることで、より良い結果を得られる。この節の残りはかなり技術的になる。せっかちな読者は次節へ進んでも構わない。ここでの内容は後続の概念の前提ではないからである。

経験リスクとリスク

まず、モデル訓練中に実際に何が起きているのかを振り返ってみよう。訓練データ $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ の特徴量と対応するラベルを順に見て、ミニバッチごとにモデル f のパラメータを更新する。簡単のため正則化は考えず、主として訓練データ上の損失を最小化する：

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(f(\mathbf{x}_i), y_i), \tag{7.7.1}$$

ここで l は、対応するラベル y_i が与えられたときの予測 $f(\mathbf{x}_i)$ が「どれだけ悪いか」を測る損失関数である。統計学では (7.7.1) の項を 経験リスクと呼ぶ。経験リスクは、真の分布 $p(\mathbf{x}, y)$ から得られるデータ全体にわたる損失の期待値である リスクを近似するための、訓練データ上の平均損失です：

$$E_{p(\mathbf{x},y)}[l(f(\mathbf{x}), y)] = \int\int l(f(\mathbf{x}), y)p(\mathbf{x}, y)\ d\mathbf{x}dy. \tag{7.7.2}$$

しかし実際には、通常データ全体を入手することはできない。したがって、経験リスク最小化、すなわち (7.7.1) の経験リスクを最小化することは、リスクを近似的に最小化することを期待した機械学習の実践的戦略である。

共変量シフトの補正

ラベル付きデータ $(\mathbf{x}_i, y_i)$ があり、ある依存関係 $P(y \mid \mathbf{x})$ を推定したいとする。残念ながら、観測 $\mathbf{x}_i$ は ソース分布 $q(\mathbf{x})$ から得られており、ターゲット分布 $p(\mathbf{x})$ からではない。幸いなことに、依存関係の仮定により条件付き分布は変化しないので、$p(y \mid \mathbf{x}) = q(y \mid \mathbf{x})$ である。ソース分布 $q(\mathbf{x})$ が「間違っている」なら、リスクにおける次の単純な恒等式を使って補正できる：

$$\int\int l(f(\mathbf{x}), y)p(y \mid \mathbf{x})p(\mathbf{x})\ d\mathbf{x}dy = \int\int l(f(\mathbf{x}), y)q(y \mid \mathbf{x})q(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})}\ d\mathbf{x}dy. \tag{7.7.3}$$

言い換えると、各データ例を、正しい分布からサンプリングされる確率と誤った分布からサンプリングされる確率の比で再重み付けする必要がある：

$$\beta_i \stackrel{\text{def}}{=} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)}. \tag{7.7.4}$$

各データ例 $(\mathbf{x}_i, y_i)$ に重み β_i を代入すれば、重み付き経験リスク最小化を用いてモデルを学習できる：

$$\underset{f}{\text{minimize}}\ \frac{1}{n}\sum_{i=1}^{n}\beta_i l(f(\mathbf{x}_i), y_i). \tag{7.7.5}$$

残念ながら、この比は分からない。したがって、何か有用なことをする前に、それを推定する必要がある。さまざまな方法が利用可能であり、最小ノルム原理や最大エントロピー原理を用いて期待演算子そのものを再較正しようとする洗練された作用素論的アプローチもある。このような手法では、両方の分布からサンプルが必要であることに注意されたい　「真の」p、たとえばテストデータへのアクセス、および訓練セット生成に使った q（後者は自明に利用可能）である。ただし、必要なのは特徴量 $\mathbf{x} \sim p(\mathbf{x})$ だけであり、ラベル $y \sim p(y)$ にアクセスする必要はない。

この場合、元の方法とほぼ同等の結果を与える非常に効果的なアプローチがある。それはロジスティック回帰である。これは二値分類におけるソフトマックス回帰の特殊例です（Section 7.1 を参照）。これだけで推定確率比を計算できる。私たちは、$p(\mathbf{x})$ から得られたデータと $q(\mathbf{x})$ から得られたデータを区別する分類器を学習する。もし 2 つの分布を区別できないなら、それは対応するイン

スタンスがどちらの分布から来たとしても同じくらい起こりやすいことを意味する。一方、うまく識別できるインスタンスは、それに応じて大きく重み付けまたは小さく重み付けすべきである。

簡単のため、両方の分布 $p(\mathbf{x})$ と $q(\mathbf{x})$ からのインスタンス数が等しいと仮定する。ここで、z を p から得られたデータには 1、q から得られたデータには-1 のラベルとする。すると、混合データセットにおける確率は

$$P(z = 1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x}) + q(\mathbf{x})} \text{ and hence } \frac{P(z = 1 \mid \mathbf{x})}{P(z = -1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}. \tag{7.7.6}$$

したがって、ロジスティック回帰アプローチを使い、$P(z = 1 \mid \mathbf{x}) = \frac{1}{1+\exp(-h(\mathbf{x}))}$（$h$ はパラメータ化された関数）とすると、次が成り立ちる。

$$\beta_i = \frac{1/(1 + \exp(-h(\mathbf{x}_i)))}{\exp(-h(\mathbf{x}_i))/(1 + \exp(-h(\mathbf{x}_i)))} = \exp(h(\mathbf{x}_i)). \tag{7.7.7}$$

その結果、2 つの問題を解く必要がある。1 つ目は、両方の分布から得られたデータを区別すること、2 つ目は、(7.7.5) における重み β_i を用いた重み付き経験リスク最小化問題である。

これで補正アルゴリズムを説明する準備が整った。訓練セット $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ とラベルなしテストセット $\{\mathbf{u}_1, \ldots, \mathbf{u}_m\}$ があるとする。共変量シフトでは、すべての $1 \le i \le n$ に対する $\mathbf{x}_i$ はあるソース分布から得られ、すべての $1 \le i \le m$ に対する $\mathbf{u}_i$ はターゲット分布から得られると仮定する。共変量シフトを補正する典型的なアルゴリズムは次のとおりである。

1. 二値分類の訓練セットを作る：$\{(\mathbf{x}_1, -1), \ldots, (\mathbf{x}_n, -1), (\mathbf{u}_1, 1), \ldots, (\mathbf{u}_m, 1)\}$。
2. ロジスティック回帰を用いて二値分類器を学習し、関数 h を得る。
3. 訓練データを $\beta_i = \exp(h(\mathbf{x}_i))$、あるいはより良くは定数 c に対して $\beta_i = \min(\exp(h(\mathbf{x}_i)), c)$ で重み付けする。
4. (7.7.5) において、$\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ で学習する際に重み β_i を使う。

上のアルゴリズムは重要な仮定に依存していることに注意されたい。この方式が機能するには、ターゲット（たとえばテスト時）の分布における各データ例が、訓練時に発生する非ゼロ確率を持っていなければならない。もし $p(\mathbf{x}) > 0$ だが $q(\mathbf{x}) = 0$ となる点があれば、対応する重要度重みは無限大になるはずである。

ラベルシフトの補正

k 個のカテゴリを持つ分類タスクを扱っているとする。Section 7.7.3 と同じ記法を用い、q と p はそれぞれソース分布（たとえば訓練時）とターゲット分布（たとえばテスト時）である。ラベルの分布が時間とともに変化し、$q(y) \ne p(y)$ だが、クラス条件付き分布は $q(\mathbf{x} \mid y) = p(\mathbf{x} \mid y)$ として同じままだと仮定する。ソース分布 $q(y)$ が「間違っている」なら、(7.7.2) で定義されたリスクにおける次

の恒等式に従って補正できる：

$$\int\int l(f(\mathbf{x}), y)p(\mathbf{x} \mid y)p(y)\, d\mathbf{x}dy = \int\int l(f(\mathbf{x}), y)q(\mathbf{x} \mid y)q(y)\frac{p(y)}{q(y)}\, d\mathbf{x}dy. \tag{7.7.8}$$

ここでの重要度重みは、ラベルの尤度比に対応する：

$$\beta_i \stackrel{\text{def}}{=} \frac{p(y_i)}{q(y_i)}. \tag{7.7.9}$$

ラベルシフトのよい点の一つは、ソース分布上でそこそこ良いモデルがあれば、周囲次元を扱うことなくこれらの重みを一貫して推定できることである。深層学習では、入力は画像のような高次元オブジェクトになりがちであるが、ラベルはカテゴリのようなより単純なオブジェクトであることが多いである。

ターゲットのラベル分布を推定するには、まず、そこそこ良い既製の分類器（通常は訓練データで学習されたもの）を取り、検証セット（これも訓練分布から得られる）を使ってその「混同行列」を計算する。混同行列 $\mathbf{C}$ は単なる $k \times k$ 行列で、各列がラベルカテゴリ（正解）に対応し、各行がモデルの予測カテゴリに対応する。各セルの値 c_{ij} は、検証セット上の全予測のうち、真のラベルが j でモデルが i を予測した割合である。

しかし、ターゲットデータ上で混同行列を直接計算することはできない。なぜなら、実運用で目にするデータ例のラベルは見えないからである。複雑なリアルタイム注釈パイプラインに投資しない限りはそうである。ただし、できることは、テスト時のモデル予測をすべて平均し、平均モデル出力 $\mu(\hat{\mathbf{y}}) \in \mathbb{R}^k$ を得ることである。ここで i^{th} 要素 $\mu(\hat{y}_i)$ は、テストセット上の全予測のうち、モデルが i を予測した割合である。

いくつかの穏やかな条件の下では　分類器が最初からそこそこ正確であり、ターゲットデータに以前見たことのあるカテゴリしか含まれず、しかもラベルシフトの仮定が最初から成り立っている（ここではこれが最も強い仮定です）　次の単純な線形方程式を解くことでテストセットのラベル分布を推定できる。

$$\mathbf{C}p(\mathbf{y}) = \mu(\hat{\mathbf{y}}), \tag{7.7.10}$$

なぜなら推定として、すべての $1 \le i \le k$ について $\sum_{j=1}^{k} c_{ij}p(y_j) = \mu(\hat{y}_i)$ が成り立つからである。ここで $p(y_j)$ は、k 次元のラベル分布ベクトル $p(\mathbf{y})$ の j^{th} 要素である。分類器が最初から十分に正確であれば、混同行列 $\mathbf{C}$ は可逆となり、$p(\mathbf{y}) = \mathbf{C}^{-1}\mu(\hat{\mathbf{y}})$ という解が得られる。

ソースデータではラベルが観測できるので、分布 $q(y)$ を推定するのは容易である。すると、ラベル y_i を持つ任意の訓練データ例 i について、推定した $p(y_i)/q(y_i)$ の比を取り、重み β_i を計算し、(7.7.5) の重み付き経験リスク最小化に代入できる。

概念シフトの補正

概念シフトを原理に基づいて修正するのは、はるかに難しいである。たとえば、突然問題が猫と犬の区別から白い動物と黒い動物の区別に変わるような状況では、新しいラベルを集めてゼロから学習し直す以上に良いことができると仮定するのは無理がある。幸い、実際にはそのような極端なシフトはまれである。むしろ通常は、タスクがゆっくり変化し続ける。より具体的にするため、いくつか例を挙げる。

- 計算広告では新製品が発売され、旧製品は人気が下がる。つまり、広告とその人気の分布は徐々に変化し、それに合わせてクリック率予測器も徐々に変化する必要がある。
- 交通監視カメラのレンズは環境による摩耗で徐々に劣化し、画像品質に段階的に影響する。
- ニュースの内容は徐々に変化する（つまり、ほとんどのニュースは変わらないが、新しい話題が現れる）。

このような場合には、データの変化に適応させるためにネットワークを訓練するときと同じアプローチを使える。言い換えると、既存のネットワーク重みを使い、ゼロから訓練するのではなく、新しいデータで数回更新するだけである。

7.7.4 学習問題の分類

分布の変化にどう対処するかを理解したので、機械学習問題の定式化に関する他の側面も考えてみよう。

バッチ学習

バッチ学習では、訓練特徴量とラベル $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ にアクセスでき、それを使ってモデル $f(\mathbf{x})$ を訓練する。その後、このモデルを同じ分布から得られた新しいデータ $(\mathbf{x}, y)$ のスコア付けに使う。ここで述べる問題のデフォルトの仮定はこれである。たとえば、猫と犬のたくさんの画像に基づいて猫検出器を訓練するかもしれない。一度訓練したら、それを賢い猫ドアのコンピュータビジョンシステムの一部として出荷し、猫だけを通すようにする。その後は顧客の家に設置され、（極端な事情がない限り）二度と更新されない。

オンライン学習

今度は、データ $(\mathbf{x}_i, y_i)$ が 1 サンプルずつ到着すると想像しよう。より具体的には、まず $\mathbf{x}_i$ を観測し、その後で $f(\mathbf{x}_i)$ の推定値を出す必要があるとする。これを行って初めて y_i を観測し、その決定に応じて報酬を得るか損失を被りる。多くの現実問題はこのカテゴリに入りる。たとえば、明日の株価を予測し、その推定に基づいて取引し、一日の終わりにその推定が利益を生んだかどうかを知る必要がある。言い換えると、オンライン学習では、新しい観測に応じてモデ

ルを継続的に改善する次のサイクルがある：

$$\begin{aligned}&\text{model } f_t \longrightarrow \text{data } \mathbf{x}_t \longrightarrow \text{estimate } f_t(\mathbf{x}_t) \longrightarrow \\ &\text{observation } y_t \longrightarrow \text{loss } l(y_t, f_t(\mathbf{x}_t)) \longrightarrow \text{model } f_{t+1}\end{aligned} \tag{7.7.11}$$

バンディット

バンディットは上の問題の特殊ケースである。多くの学習問題では、パラメータを学習したい連続的にパラメータ化された関数 f（たとえば深層ネットワーク）を扱いるが、バンディット問題では、引ける腕の数が有限、すなわち取れる行動の数が有限である。このより単純な問題では、最適性に関してより強い理論保証が得られるのはそれほど驚くことではない。ここで取り上げるのは主に、この問題がしばしば（紛らわしく）別個の学習設定として扱われるからである。

制御

多くの場合、環境は私たちが何をしたかを覚えている。必ずしも敵対的というわけではないが、単に記憶しており、応答は以前に何が起きたかに依存する。たとえば、コーヒーボイラーの制御装置は、以前にボイラーを加熱していたかどうかによって異なる温度を観測する。PID（比例・積分・微分）制御アルゴリズムはそこでよく使われる。同様に、ニュースサイトでのユーザーの行動は、以前に何を見せたかに依存する（たとえば、多くのニュースは一度しか読まれない）。このようなアルゴリズムの多くは、自分が作用する環境のモデルを作り、意思決定がよりランダムに見えないようにする。近年では、制御理論（たとえば PID の変種）が、より良い分離表現と再構成品質を得るためのハイパーパラメータの自動調整や、生成テキストの多様性および生成画像の再構成品質の改善にも使われている (Shao *et al.*, 2020)。

強化学習

記憶を持つ環境というより一般的な場合には、環境が私たちと協力しようとする状況（協力ゲーム、特にゼロ和でないゲーム）や、逆に環境が勝とうとする状況に出会うかもしれない。チェス、囲碁、バックギャモン、StarCraft などは強化学習の例である。同様に、自動運転車のための良い制御器を作りたいこともある。他の車は、自動運転車の運転スタイルに対して非自明な形で反応する可能性が高く、たとえば避けようとしたり、事故を起こそうとしたり、協力しようとしたりする。

環境を考慮する

上のさまざまな状況の重要な違いの一つは、静的な環境では通用した戦略が、適応可能な環境では通用しないかもしれないことである。たとえば、トレーダーが見つけた裁定機会は、一度利用されると消えてしまう可能性が高いである。環境が変化する速度とその仕方が、私たちが適用できるアルゴリズムの種

類を大きく左右する。たとえば、変化がゆっくりしか起こらないと分かっていれば、推定値もゆっくりしか変化しないようにできる。環境が瞬時に変化しうるが、それが非常にまれだと分かっていれば、その点を考慮できる。こうした知識は、解かれている問題が時間とともに変化しうる概念シフトに対処するうえで、志あるデータサイエンティストにとって極めて重要である。

7.7.5 機械学習における公平性、説明責任、透明性

最後に、機械学習システムを導入するとき、単に予測モデルを最適化しているだけではないことを忘れてはならない　通常は、意思決定を（部分的または完全に）自動化するために使われる道具を提供しているのである。こうした技術システムは、その結果として下される決定の対象となる個人の生活に影響を与える。予測を考えることから意思決定へと飛躍すると、新しい技術的問題だけでなく、慎重に検討すべき倫理的問題が大量に生じる。医療診断システムを導入するなら、どの集団に対して有効で、どの集団では有効でないのかを知る必要がある。ある集団に対する予見可能なリスクを見落とすと、劣った医療を提供してしまうかもしれない。さらに、意思決定システムを考えるときには、技術の評価方法を一歩引いて見直す必要がある。この視点の変化に伴う多くの帰結の中でも、精度がめったに適切な指標ではないことが分かりる。たとえば、予測を行動に変換するときには、さまざまな誤り方のコスト感度を考慮したくなることが多いである。ある種類の画像誤分類が人種的なごまかしと受け取られうる一方で、別のカテゴリへの誤分類は無害だとしたら、社会的価値を反映して意思決定プロトコルの設計における閾値を調整したくなるだろう。また、予測システムがフィードバックループを生みうる点にも注意が必要である。たとえば、予測的警備システムを考えてみよう。これは、予測された犯罪率が高い地域に警察官の巡回を割り当てるものである。次のような懸念すべきパターンが生じるのは容易に想像できる。

1. 犯罪の多い地域にはより多くの巡回が行われる。
2. その結果、これらの地域でより多くの犯罪が発見され、将来の反復で利用可能な訓練データに入る。
3. より多くの陽性例にさらされることで、モデルはこれらの地域でさらに多くの犯罪を予測する。
4. 次の反復では、更新されたモデルが同じ地域をさらに強く標的にし、さらに多くの犯罪が発見される、という具合に続く。

しばしば、モデルの予測が訓練データと結びつくさまざまな仕組みは、モデリング過程で考慮されていない。これは研究者が *runaway feedback loops* と呼ぶものにつながる可能性がある。さらに、そもそも正しい問題を解いているのかどうかにも注意を払う必要がある。予測アルゴリズムは今や、情報の流通を媒介するうえで非常に大きな役割を果たしている。個人が目にするニュースは、その人が *Like* した Facebook ページの集合によって決まるべきだろうか？これらは、機械学習のキャリアで遭遇しうる数多くの差し迫った倫理的ジレンマのほんの一部にすぎない。

7.7.6 まとめ

多くの場合、訓練セットとテストセットは同じ分布から来ない。これは分布シフトと呼ばれる。リスクとは、真の分布から得られるデータ全体にわたる損失の期待値である。しかし、このデータ全体は通常利用できない。経験リスクは、リスクを近似するための訓練データ上の平均損失である。実際には、経験リスク最小化を行う。

対応する仮定の下では、共変量シフトとラベルシフトはテスト時に検出・補正できる。この偏りを考慮しないと、テスト時に問題が生じる可能性がある。場合によっては、環境が自動化された行動を記憶し、予想外の形で応答することがある。モデルを構築するときにはこの可能性を考慮し、ライブシステムを継続的に監視して、モデルと環境が予期しない形で絡み合う可能性を受け入れる必要がある。

7.7.7 演習

1. 検索エンジンの挙動を変えると何が起こりうるだろうか。ユーザーはどうするだろうか。広告主はどうだろうか。
2. 共変量シフト検出器を実装せよ。ヒント：分類器を作る。
3. 共変量シフト補正器を実装せよ。
4. 分布シフト以外に、経験リスクがリスクをどの程度近似するかに影響しうるものは何だろうか。

8 多層パーセプトロン

この章では、あなたにとって最初の本当の 深いネットワークを紹介する。
最も単純な深層ネットワークは 多層パーセプトロンと呼ばれ、複数の層のニューロンから構成される。各層は、その下の層（そこから入力を受け取る）およびその上の層（そこに影響を与える）にあるニューロンと完全に結合されている。
自動微分は深層学習アルゴリズムの実装を大幅に簡単にするが、深層ネットワークにおいてこれらの勾配がどのように計算されるのかを深く掘り下げていくる。
その後、数値安定性とパラメータ初期化に関する問題について議論できるようになる。これらは深層ネットワークをうまく学習させるための鍵となる。
このような高容量モデルを学習すると、過学習のリスクがある。したがって、深層ネットワークにおける正則化と汎化を改めて取り上げる。
全体を通して、概念だけでなく、深層ネットワークを使う実践についてもしっかり理解できるようにすることを目指する。
この章の最後では、ここまでに導入した内容を実際のケース、すなわち住宅価格予測に適用する。計算性能、スケーラビリティ、効率性に関する問題は、後続の章に先送りする。

8.1 多層パーセプトロン

Section 7.1 では、ソフトマックス回帰を導入し、アルゴリズムをスクラッチから実装し（Section 7.4）、高水準 API を用いて実装した（Section 7.5）。これにより、低解像度画像から 10 種類の衣類を認識できる分類器を学習できた。その過程で、データを整形する方法、出力を妥当な確率分布に変換する方法、適切な損失関数を適用する方法、そしてモデルのパラメータに関してそれを最小化する方法を学んだ。単純な線形モデルの文脈でこれらの仕組みを習得した今、いよいよ深層ニューラルネットワークの探究を始めることができる。本書が主として扱うのは、より豊かなクラスに属するこれらのモデルである。

```
%matplotlib inline
from d2l import torch as d2l
import torch
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, np, npx
npx.set_np()
```

```
%matplotlib inline
from d2l import jax as d2l
import jax
from jax import numpy as jnp
from jax import grad, vmap
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf
```

8.1.1 隠れ層

Section 6.1.1 では、アフィン変換をバイアスを加えた線形変換として説明した。まず、Fig. 7.1.1 に示したソフトマックス回帰の例に対応するモデル構造を思い出ししよう。このモデルは、単一のアフィン変換の後にソフトマックス演算を適用することで、入力を直接出力へ写像する。もしラベルが本当に単純なアフィン変換によって入力データと結び付いているのであれば、この方法で十分である。しかし、線形性（アフィン変換における）は 強い仮定である。

線形モデルの限界

たとえば、線形性は 単調性という、より 弱い仮定を含意する。すなわち、特徴量が増加したとき、モデルの出力は必ず増加するか（対応する重みが正の場合）、あるいは必ず減少するか（対応する重みが負の場合）のどちらかでなければなりない。これは場合によっては理にかなっている。たとえば、ある個人がローンを返済するかどうかを予測したいとししよう。他の条件が同じなら、収入が高い申請者のほうが低い申請者よりも返済する可能性が常に高い、と仮定するのは妥当かもしれない。この関係は単調ではあるが、返済確率と線形に結び付いているとはおそらく言えない。収入が $0 から $50,000 に増えることは、$1 million から $1.05 million に増えることよりも、返済可能性を大きく高めるだろう。この問題への一つの対処法は、ロジスティック写像（したがって結果の確率の対数）を用いて、線形性がよりもっともらしくなるように出力を後処理することである。

単調性すら破る例も簡単に思いつくる。たとえば、体温の関数として健康状態を予測したいとししよう。37°C（98.6°F）を超える正常体温の個人では、体温が高いほどリスクが高いことを示する。しかし、体温が 37°C を下回ると、今度は体温が低いほどリスクが高いのです！ここでも、37°C からの距離を特徴量として使うなど、何らかの巧妙な前処理で問題を解決できるかもしれない。

では、猫と犬の画像分類はどうだろうか？位置 (13, 17) にある画素の強度を増やすと、その画像が犬である確率は常に増える（あるいは常に減る）べきだろうか？線形モデルに依存することは、猫と犬を区別するために必要なのは個々の画素の明るさを評価することだけだ、という暗黙の仮定に対応する。画像を反転してもカテゴリが保存されるような世界では、この方法は失敗する運命にある。

それでも、ここでの線形性は前の例と比べて一見ばかげているにもかかわらず、単純な前処理の修正で問題を解決できるかはそれほど明らかではありない。というのも、任意の画素の意味はその文脈（周囲の画素の値）に複雑に依存するからである。特徴量間の関連する相互作用を考慮した表現が存在し、その上で線形モデルが適切になる可能性はあるが、それを手作業で計算する方法は分かりない。深層ニューラルネットワークでは、観測データを用いて、隠れ層による表現と、その表現に作用する線形予測器の両方を同時に学習した。

この非線形性の問題は、少なくとも 1 世紀にわたって研究されてきた (Fisher, 1925)。たとえば、決定木は最も基本的な形では、一連の二値決定を用いてクラス所属を判定する (Quinlan, 1993)。同様に、カーネル法は非線形依存をモデル化するために何十年も使われてきた (Aronszajn, 1950)。これは非パラメトリックなスプラインモデル (Wahba, 1990) やカーネル法 (Schölkopf and Smola, 2002) にもつながっている。また、脳はこれをごく自然に解いている。結局のところ、ニューロンは他のニューロンへ入力し、そのニューロンはさらに別のニューロンへ入力する (Ramón y Cajal and Azoulay, 1894)。その結果、比較的単純な変換が連なった構造になる。

隠れ層の導入

1 つ以上の隠れ層を組み込むことで、線形モデルの限界を克服できる。最も簡単な方法は、多数の全結合層を順に積み重ねることである。各層はその上の層へ入力を渡し、最終的に出力を生成する。最初の $L-1$ 層を表現として、最後の層を線形予測器として考えることができる。この構造は一般に 多層パーセプトロンと呼ばれ、しばしば *MLP* と略される（Fig. 8.1.1）。

この MLP は 4 つの入力、3 つの出力を持ち、隠れ層には 5 個の隠れユニットがある。入力層には計算が含まれないため、このネットワークで出力を生成するには隠れ層と出力層の両方の計算を実装する必要がある。したがって、この MLP の層数は 2 である。なお、どちらの層も全結合である。すべての入力は隠れ層のすべてのニューロンに影響し、それぞれのニューロンはさらに出力層のすべてのニューロンに影響する。とはいえ、まだ終わりではありない。

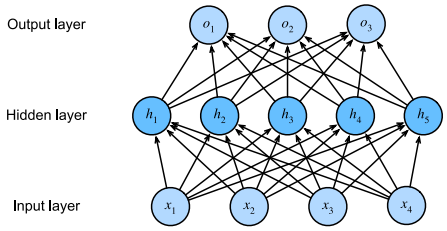


Fig. 8.1.1 5 個の隠れユニットを持つ隠れ層を備えた MLP。

線形から非線形へ

これまでと同様に、$\mathbf{X} \in \mathbb{R}^{n \times d}$ を、各データ例が d 個の入力（特徴量）を持つ n 個のデータ例からなるミニバッチを表す行列とする。隠れ層が h 個の隠れユニットを持つ 1 隠れ層 MLP について、$\mathbf{H} \in \mathbb{R}^{n \times h}$ を隠れ層の出力、すなわち 隠れ表現とする。隠れ層と出力層はいずれも全結合なので、隠れ層の重み $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ とバイアス $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$、出力層の重み $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ とバイアス $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$ を持つ。これにより、1 隠れ層 MLP の出力 $\mathbf{O} \in \mathbb{R}^{n \times q}$ を次のように計算できる。

$$\begin{aligned} \mathbf{H} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{8.1.1}$$

隠れ層を追加すると、モデルは新たに追加されたパラメータ群を追跡し更新する必要がある。では、その代わりに何を得たのだろうか？驚くかもしれないが、上で定義したモデルでは、苦労のわりに何も得ていない！理由は明白である。上の隠れユニットは入力のアフィン関数で与えられ、出力（ソフトマックス適用前）は隠れユニットのアフィン関数にすぎない。アフィン関数のアフィン関数は、やはりアフィン関数である。さらに、線形モデルはすでに任意のアフィン関数を表現できた。

これを形式的に見るには、上の定義から隠れ層を消去すればよく、パラメータ $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ と $\mathbf{b} = \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$ を持つ等価な単層モデルが得られる。

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}. \tag{8.1.2}$$

多層構造の潜在能力を引き出すには、もう 1 つ重要な要素が必要である。それは、アフィン変換の後に各隠れユニットへ適用する非線形 活性化関数 σ である。たとえば、よく使われる選択肢は ReLU（rectified linear unit）活性化関数 (Nair and Hinton, 2010) $\sigma(x) = \max(0, x)$ で、引数に要素ごとに作用する。活性化関数 $\sigma(\cdot)$ の出力は 活性化と呼ばれる。一般に、活性化関数を導入すると、MLP を線形モデルへ畳み込むことはもはやできない。

$$\begin{aligned} \mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{8.1.3}$$

$\mathbf{X}$ の各行はミニバッチ中の 1 つのデータ例に対応するので、記法を少し乱用

して、非線形性 σ は入力に行ごとに作用する、すなわち 1 データ例ずつ適用されると定義する。Section 7.1.1 で行ごとの演算を表す際にソフトマックスでも同じ記法を使ったことに注意されたい。私たちが使う活性化関数は、行ごとだけでなく要素ごとに作用することが非常に多いである。つまり、層の線形部分を計算した後は、他の隠れユニットの値を見なくても各活性化を計算できる。

より一般的な MLP を構築するには、このような隠れ層をさらに積み重ねればよいである。たとえば、$\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$、$\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$ のように、層を次々と重ねることで、ますます表現力の高いモデルが得られる。

万能近似器

脳が非常に高度な統計解析を行えることは分かっている。したがって、深いネットワークがどれほど強力になり得るのかを問う価値がある。この問いには複数回答えが与えられており、たとえば MLP の文脈では Cybenko (1989)、再生核ヒルベルト空間の文脈では Micchelli (1984) があり、これは 1 つの隠れ層を持つ RBF（radial basis function）ネットワークと見なせる形である。これらの結果（および関連する結果）は、たとえ 1 隠れ層ネットワークであっても、十分な数のノード（おそらく途方もなく多く）と適切な重みがあれば、任意の関数をモデル化できることを示唆している。ただし、その関数を実際に学習することが難しいのである。ニューラルネットワークは、C 言語のようなものだと考えるとよいだろう。この言語は、他の現代的な言語と同様に、計算可能なプログラムなら何でも表現できる。しかし、仕様を満たすプログラムを実際に作ることが難しいのである。

さらに、1 隠れ層ネットワークが 任意の関数を学習できるからといって、すべての問題をそれ 1 つで解くべきだという意味ではありない。実際、この場合はカーネル法のほうがはるかに有効である。なぜなら、無限次元空間であっても問題を 厳密に解けるからです (Kimeldorf and Wahba, 1971, Schölkopf *et al.*, 2001)。実際、より深い（より広いのではなく）ネットワークを使うことで、多くの関数をはるかにコンパクトに近似できる (Simonyan and Zisserman, 2014)。より厳密な議論は後続の章で扱いる。

8.1.2 活性化関数

活性化関数は、重み付き和を計算し、さらにバイアスを加えることで、ニューロンを活性化するかどうかを決定する。それらは入力信号を出力へ変換する微分可能な演算子であり、その多くは非線形性を付加する。活性化関数は深層学習の基礎なので、代表的なものを簡単に見ておこう。

ReLU 関数

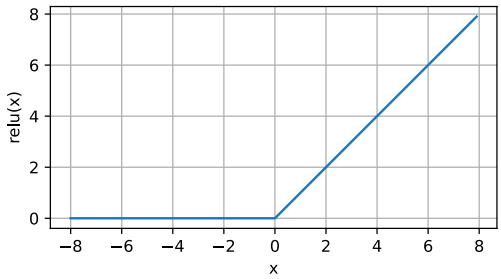
実装が簡単で、さまざまな予測タスクで良好な性能を示すことから、最も人気のある選択肢は *rectified linear unit*（*ReLU*）です (Nair and Hinton, 2010)。ReLU は

非常に単純な非線形変換を提供する。要素 x に対して、この関数はその要素と 0 の最大値として定義される。

$$\mathrm{ReLU}(x) = \max(x, 0). \tag{8.1.4}$$

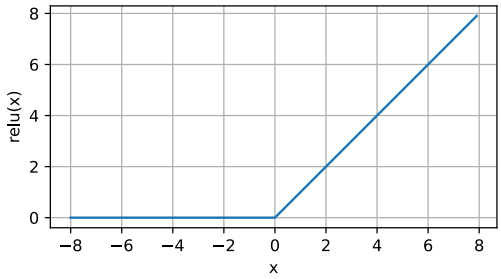
直感的には、ReLU 関数は正の要素だけを残し、対応する活性化を 0 にすることで負の要素をすべて捨てる。直感を得るために、この関数をプロットしてみしよう。見てのとおり、活性化関数は区分線形である。

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```

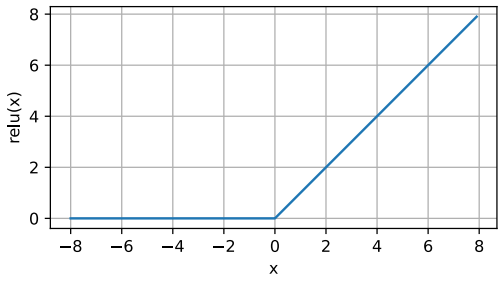



```
x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.relu(x)
d2l.plot(x, y, 'x', 'relu(x)', figsize=(5, 2.5))
```

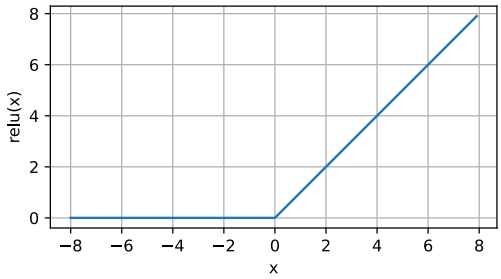
```
[07:04:24] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```




```
x = jnp.arange(-8.0, 8.0, 0.1)
y = jax.nn.relu(x)
d2l.plot(x, y, 'x', 'relu(x)', figsize=(5, 2.5))
```




```
x = tf.Variable(tf.range(-8.0, 8.0, 0.1), dtype=tf.float32)
y = tf.nn.relu(x)
d2l.plot(x.numpy(), y.numpy(), 'x', 'relu(x)', figsize=(5, 2.5))
```

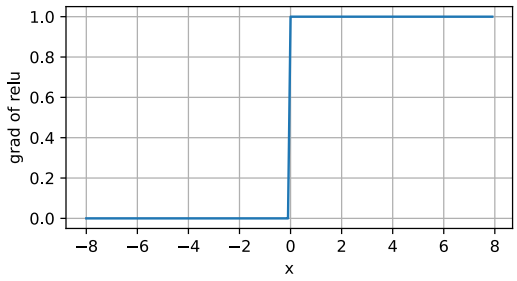


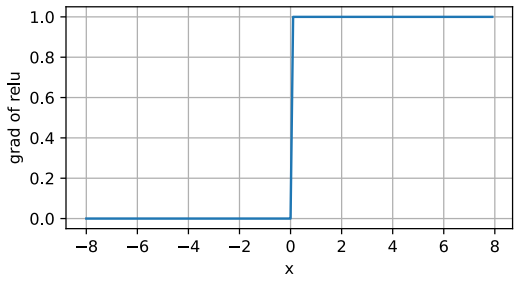
入力が負のとき、ReLU 関数の導関数は 0 であり、入力が正のとき、ReLU 関数の導関数は 1 である。ReLU 関数は、入力がちょうど 0 のときには微分可能でないことに注意されたい。この場合は左側微分を採用し、入力が 0 のときの導関数は 0 とする。これは、入力が実際には 0 にならないかもしれないので問題ありない（数学者は、これは測度 0 の集合上で非微分可能だと言うだろう）。微妙な境界条件が重要なら、私たちはおそらく工学ではなく（本当の）数学をしているのだ、という古い格言がある。ここでもその常識が当てはまるかもしれないし、少なくとも制約付き最適化を行っていないという事実が関係しているのかもしれない (Mangasarian, 1965, Rockafellar, 1970)。以下に ReLU 関数の導関数をプロットする。

```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```

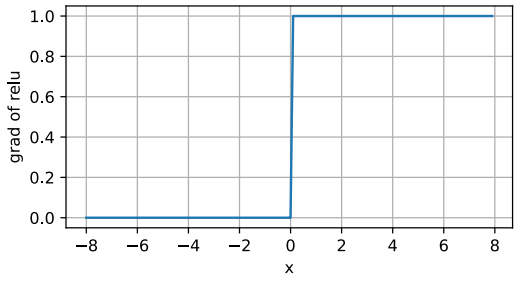
```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```

```
[07:04:24] ../src/base.cc:48: GPU context requested, but no GPUs found.
```





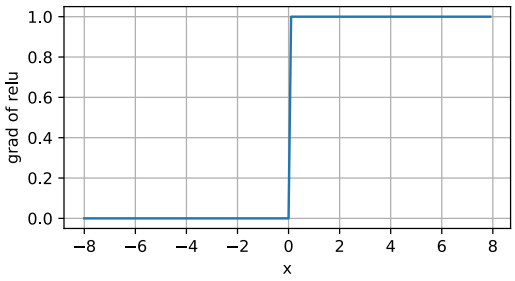

```
grad_relu = vmap(grad(jax.nn.relu))
d2l.plot(x, grad_relu(x), 'x', 'grad of relu', figsize=(5, 2.5))
```




```
with tf.GradientTape() as t:
    y = tf.nn.relu(x)
d2l.plot(x.numpy(), t.gradient(y, x).numpy(), 'x', 'grad of relu',
         figsize=(5, 2.5))
```

ReLU を使う理由は、その導関数が非常に扱いやすいからである。つまり、0 になるか、引数をそのまま通すかのどちらかである。これにより最適化が扱いやすくなり、以前のニューラルネットワークを悩ませていた勾配消失問題を軽減した（これについては後で詳しく述べる）。

ReLU 関数には多くの変種があり、たとえば *parametrized ReLU*（*pReLU*）関数 (He *et al.*, 2015) がある。この変種は ReLU に線形項を加えるため、引数が負でも



一部の情報が通過する。

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x). \tag{8.1.5}$$

シグモイド関数

シグモイド関数は、値が $\mathbb{R}$ の範囲にある入力を区間 (0, 1) 上の出力へ変換する。そのため、シグモイドはしばしば 圧縮関数と呼ばれる。つまり、(-inf, inf) の範囲にある任意の入力を (0, 1) の範囲のある値へ押し込めるのである。

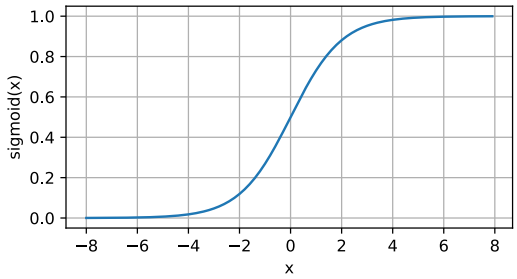
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{8.1.6}$$

初期のニューラルネットワークでは、科学者たちは生物学的ニューロンが 発火するか 発火しないかをモデル化することに関心を持っていた。そのため、この分野の先駆者たち、人工ニューロンの発明者である McCulloch と Pitts にまでさかのぼる研究では、しきい値ユニットに注目していた (McCulloch and Pitts, 1943)。しきい値型活性化は、入力があるしきい値を下回ると 0、しきい値を超えると 1 を取る。

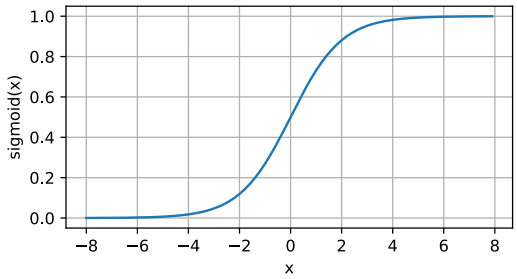
勾配ベース学習へ関心が移ると、シグモイド関数は自然な選択であった。なぜなら、それはしきい値ユニットの滑らかで微分可能な近似だからである。シグモイドは、二値分類問題で出力を確率として解釈したいときに、出力ユニットの活性化関数として今でも広く使われている。シグモイドはソフトマックスの特殊な場合と考えることができる。しかし、隠れ層での多くの用途では、シグモイドはより単純で学習しやすい ReLU にほぼ置き換えられた。その大きな理由は、シグモイドが最適化において難しさをもたらすからです (LeCun *et al.*, 1998)。大きな正の引数でも負の引数でも勾配が消えてしまうためである。その結果、抜け出しにくい平坦領域が生じることがある。それでもシグモイドは重要である。後の章（たとえば Section 13.1）で扱う再帰ニューラルネットワークでは、時間をまたいだ情報の流れを制御するためにシグモイドユニットを活用する構造を説明する。

以下にシグモイド関数をプロットする。入力が 0 に近いとき、シグモイド関数は線形変換に近づくことに注意されたい。

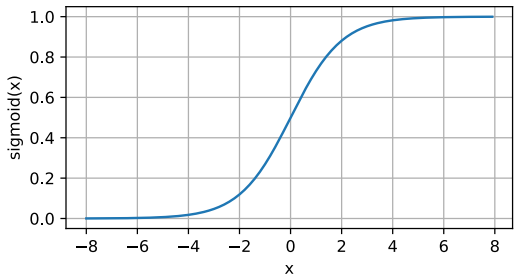
```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



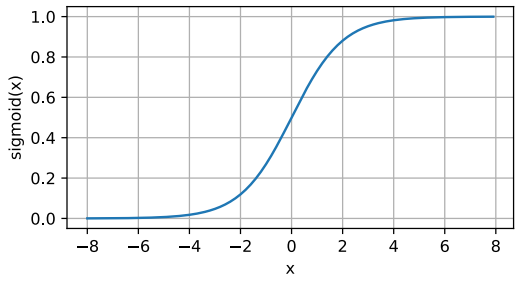

```
with autograd.record():
    y = npx.sigmoid(x)
d2l.plot(x, y, 'x', 'sigmoid(x)', figsize=(5, 2.5))
```




```
y = jax.nn.sigmoid(x)
d2l.plot(x, y, 'x', 'sigmoid(x)', figsize=(5, 2.5))
```




```
y = tf.nn.sigmoid(x)
d2l.plot(x.numpy(), y.numpy(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

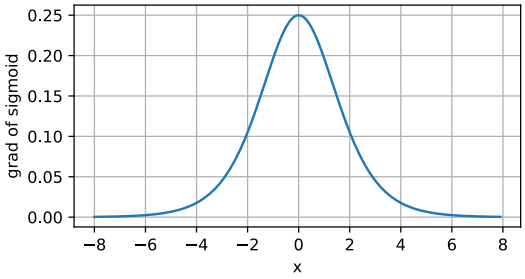


シグモイド関数の導関数は次の式で与えられる。

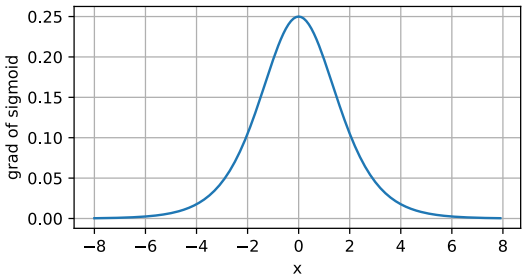
$$\frac{d}{dx}\,\mathrm{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \mathrm{sigmoid}(x)\,(1-\mathrm{sigmoid}(x))\,. \tag{8.1.7}$$

シグモイド関数の導関数を以下に示する。入力が 0 のとき、シグモイド関数の導関数は最大値 0.25 に達する。入力が 0 からどちらの方向へ離れても、導関数は 0 に近づくる。

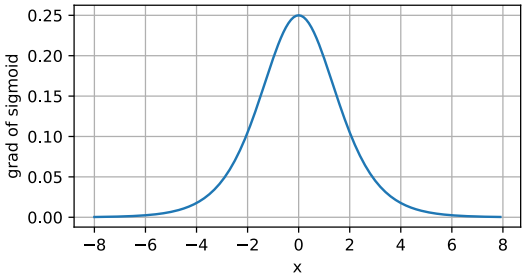
```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



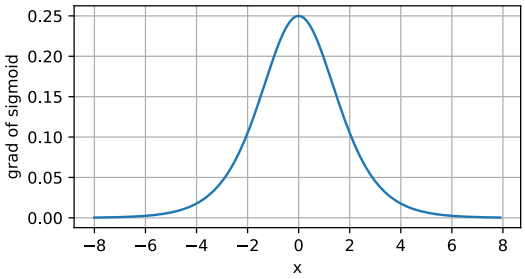

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



```
grad_sigmoid = vmap(grad(jax.nn.sigmoid))
d2l.plot(x, grad_sigmoid(x), 'x', 'grad of sigmoid', figsize=(5, 2.5))
```




```
with tf.GradientTape() as t:
    y = tf.nn.sigmoid(x)
d2l.plot(x.numpy(), t.gradient(y, x).numpy(), 'x', 'grad of sigmoid',
         figsize=(5, 2.5))
```



tanh 関数

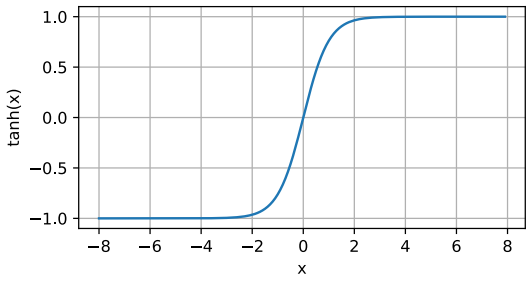
シグモイド関数と同様に、tanh（双曲線正接）関数も入力を圧縮し、それらを -1 と 1 の間の区間上の要素へ変換する。

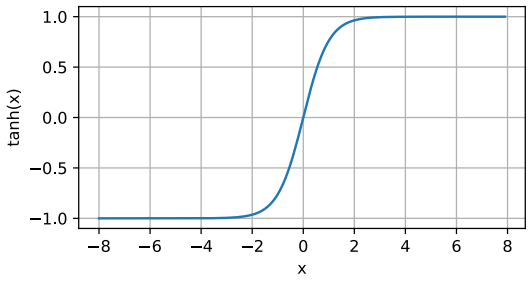
$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}. \tag{8.1.8}$$

以下に tanh 関数をプロットする。入力が 0 に近づくと、tanh 関数は線形変換に近づくことに注意されたい。関数の形はシグモイド関数に似ているが、tanh 関数は座標系の原点に関して点対称です (Kalman and Kwasny, 1992)。

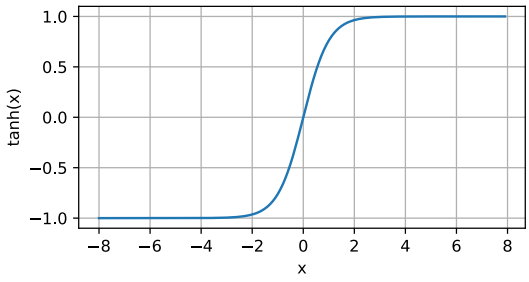
```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```

```
with autograd.record():
    y = np.tanh(x)
d2l.plot(x, y, 'x', 'tanh(x)', figsize=(5, 2.5))
```






```
y = jax.nn.tanh(x)
d2l.plot(x, y, 'x', 'tanh(x)', figsize=(5, 2.5))
```

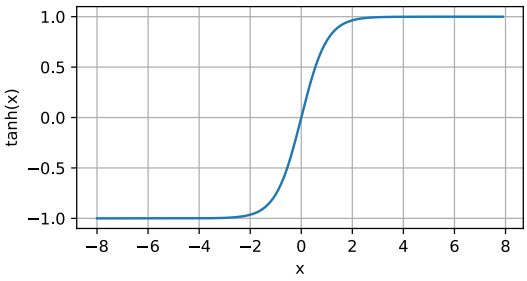



```
y = tf.nn.tanh(x)
d2l.plot(x.numpy(), y.numpy(), 'x', 'tanh(x)', figsize=(5, 2.5))
```

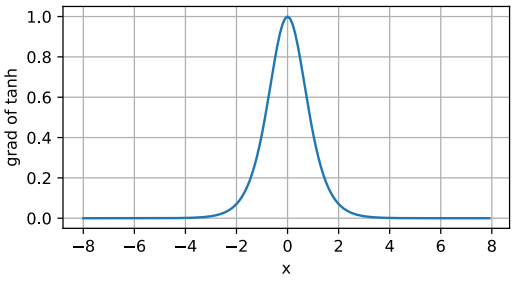
tanh 関数の導関数は次のとおりである。

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x). \tag{8.1.9}$$

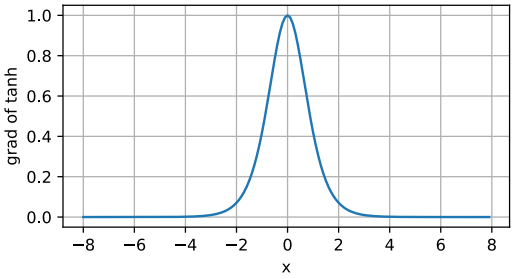
以下に示する。入力が 0 に近づくと、tanh 関数の導関数は最大値 1 に近づくる。そしてシグモイド関数で見たように、入力が 0 からどちらの方向へ離れても、tanh 関数の導関数は 0 に近づくる。




```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



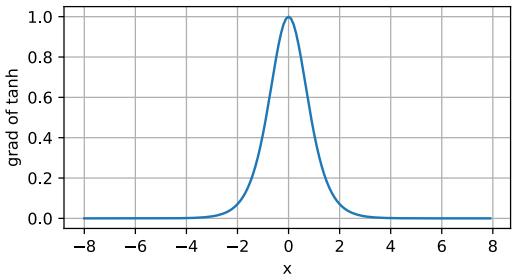

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```




```
grad_tanh = vmap(grad(jax.nn.tanh))
d2l.plot(x, grad_tanh(x), 'x', 'grad of tanh', figsize=(5, 2.5))
```

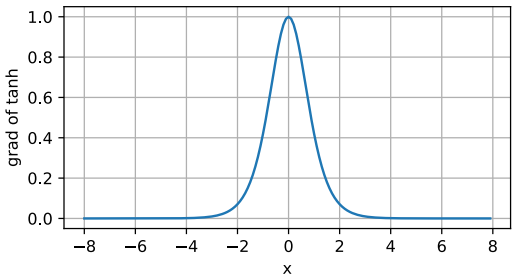
```
with tf.GradientTape() as t:
    y = tf.nn.tanh(x)
```

(continues on next page)



(continued from previous page)

```
d2l.plot(x.numpy(), t.gradient(y, x).numpy(), 'x', 'grad of tanh',
         figsize=(5, 2.5))
```



8.1.3 要約と考察

これで、非線形性を組み込んで表現力の高い多層ニューラルネットワーク構造を構築する方法が分かった。補足すると、ここまでの知識だけでも 1990 年頃の実務家と同程度の道具立てを使いこなせることになる。ある意味では、当時の誰よりも有利である。強力なオープンソースの深層学習フレームワークを活用して、わずかなコードで素早くモデルを構築できるからである。以前は、これらのネットワークを学習させるには、研究者が C、Fortran、あるいは（LeNet の場合は）Lisp で層や導関数を明示的に実装する必要があった。

副次的な利点として、ReLU はシグモイドや tanh 関数よりも最適化にかなり適している。これは、過去 10 年の深層学習の復活を支えた重要な革新の 1 つだったと言えるだろう。ただし、活性化関数の研究が止まったわけではありない。たとえば、GELU（Gaussian error linear unit）活性化関数 $x\Phi(x)$ は Hendrycks and Gimpel (2016) によるもので（$\Phi(x)$ は標準ガウス累積分布関数）、また Swish 活性化関数 $\sigma(x) = x\,\mathrm{sigmoid}(\beta x)$ は Ramachandran *et al.* (2017) により提案され、多くの場合でより高い精度をもたらす。

8.1.4 演習

1. 非線形性 σ を持たない、すなわち 線形な深層ネットワークに層を追加しても、ネットワークの表現力は決して増えないことを示しなさい。逆に、それを実際に低下させる例を挙げなさい。
2. pReLU 活性化関数の導関数を求めなさい。
3. Swish 活性化関数 $x\,\mathrm{sigmoid}(\beta x)$ の導関数を求めなさい。
4. ReLU（または pReLU）だけを用いる MLP が、連続な区分線形関数を構成することを示しなさい。
5. シグモイドと tanh は非常によく似ている。
   1. $\tanh(x) + 1 = 2\,\mathrm{sigmoid}(2x)$ を示しなさい。
   2. 両方の非線形性でパラメータ化される関数クラスが同一であることを証明しなさい。ヒント: アフィン層にもバイアス項がある。
6. バッチ正規化 (Ioffe and Szegedy, 2015) のように、1 つのミニバッチごとに作用する非線形性があると仮定する。どのような問題が生じると予想されるか？
7. シグモイド活性化関数で勾配が消失する例を示しなさい。

8.2 多層パーセプトロンの実装

多層パーセプトロン（MLP）は、単純な線形モデルよりも実装がそれほど複雑になるわけではありない。重要な概念上の違いは、複数の層を連結するようになったことである。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

8.2.1 ゼロからの実装

まずは、こうしたネットワークをゼロから実装してみしよう。

モデルパラメータの初期化

Fashion-MNIST には 10 クラスがあり、各画像は $28 \times 28 = 784$ 個のグレースケール画素値の格子から成る。これまでと同様に、ここではひとまず画素間の空間構造は無視するので、784 個の入力特徴量と 10 クラスをもつ分類データセットと考えられる。まずは、1 つの隠れ層と 256 個の隠れユニットをもつ MLP を実装ししよう。層の数もその幅も調整可能である（これらはハイパーパラメータと見なされる）。通常、層の幅は 2 の大きな冪で割り切れるように選ぶ。これは、ハードウェアにおけるメモリの割り当てとアドレス指定の仕組みにより、計算効率がよいからである。

ここでも、パラメータは複数のテンソルで表する。各層ごとに、1 つの重み行列と 1 つのバイアスベクトルを保持しなければならないことに注意されたい。いつものように、これらのパラメータに関する損失の勾配のためのメモリも確保する。

以下のコードでは `nn.Parameter` を使って、クラス属性を `autograd` によって追跡されるパラメータとして自動的に登録する（Section 5.5）。

```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens) * sigma)
        self.b1 = nn.Parameter(torch.zeros(num_hiddens))
        self.W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * sigma)
        self.b2 = nn.Parameter(torch.zeros(num_outputs))
```

```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = np.random.randn(num_inputs, num_hiddens) * sigma
        self.b1 = np.zeros(num_hiddens)
        self.W2 = np.random.randn(num_hiddens, num_outputs) * sigma
        self.b2 = np.zeros(num_outputs)
        for param in self.get_scratch_params():
            param.attach_grad()
```

```
class MLPScratch(d2l.Classifier):
    num_inputs: int
    num_outputs: int
    num_hiddens: int
    lr: float
    sigma: float = 0.01

    def setup(self):
        self.W1 = self.param('W1', nn.initializers.normal(self.sigma),
                             (self.num_inputs, self.num_hiddens))
        self.b1 = self.param('b1', nn.initializers.zeros, self.num_hiddens)
        self.W2 = self.param('W2', nn.initializers.normal(self.sigma),
                             (self.num_hiddens, self.num_outputs))
        self.b2 = self.param('b2', nn.initializers.zeros, self.num_outputs)
```

```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = tf.Variable(
            tf.random.normal((num_inputs, num_hiddens)) * sigma)
        self.b1 = tf.Variable(tf.zeros(num_hiddens))
        self.W2 = tf.Variable(
            tf.random.normal((num_hiddens, num_outputs)) * sigma)
        self.b2 = tf.Variable(tf.zeros(num_outputs))
```

モデル

すべてがどのように動くのかを確実に理解するために、組み込みの `relu` 関数を直接呼び出すのではなく、ReLU 活性化関数を自分で実装する。

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

```
def relu(X):
    return np.maximum(X, 0)
```

```
def relu(X):
    return jnp.maximum(X, 0)
```

```
def relu(X):
    return tf.math.maximum(X, 0)
```

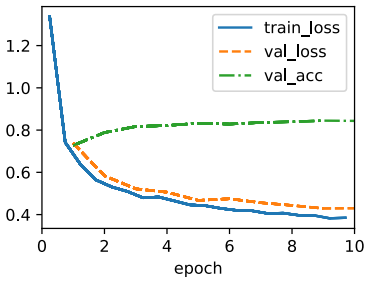
空間構造を無視するので、各 2 次元画像を `reshape` して長さ `num_inputs` の平坦なベクトルに変換する。最後に、わずか数行のコードでモデルを実装する。フレームワーク組み込みの autograd を使うので、これだけで十分である。

```
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = d2l.reshape(X, (-1, self.num_inputs))
    H = relu(d2l.matmul(X, self.W1) + self.b1)
    return d2l.matmul(H, self.W2) + self.b2
```

学習

幸いなことに、MLP の学習ループはソフトマックス回帰の場合とまったく同じである。モデル、データ、トレーナーを定義し、最後にモデルとデータに対して `fit` メソッドを呼び出する。

```
model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



8.2.2 簡潔な実装

予想どおり、高水準 API に頼れば、MLP はさらに簡潔に実装できる。

モデル

ソフトマックス回帰の簡潔な実装（Section 7.5）と比べると、違いは、以前は *1* つだけ追加していた全結合層を、ここでは *2* つ追加する点だけである。1 つ目が隠れ層で、2 つ目が出力層である。

```
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
                                 nn.ReLU(), nn.LazyLinear(num_outputs))
```

```
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential()
        self.net.add(nn.Dense(num_hiddens, activation='relu'),
                     nn.Dense(num_outputs))
        self.net.initialize()
```

```
class MLP(d2l.Classifier):
    num_outputs: int
    num_hiddens: int
    lr: float

    @nn.compact
    def __call__(self, X):
        X = X.reshape((X.shape[0], -1))  # Flatten
        X = nn.Dense(self.num_hiddens)(X)
        X = nn.relu(X)
        X = nn.Dense(self.num_outputs)(X)
        return X
```

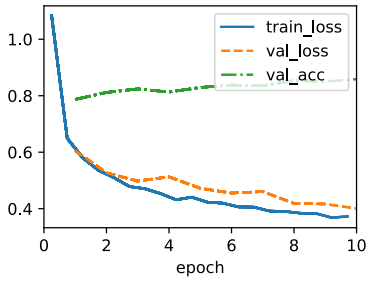
```
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = tf.keras.models.Sequential([
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(num_hiddens, activation='relu'),
            tf.keras.layers.Dense(num_outputs)])
```

以前は、モデルのパラメータを使って入力を変換する `forward` メソッドを定義していた。これらの操作は本質的にはパイプラインである。つまり、入力を受け取り、変換（たとえば、重みとの行列積の後にバイアスを加える）を適用し、その変換の出力を次の変換への入力として繰り返し使いる。しかし、ここでは `forward` メソッドが定義されていないことに気づいたかもしれない。実際には、`MLP` は `Module` クラス（Section 6.2.2）から `forward` メソッドを継承し、単に `self.net(X)`（`X` は入力）を呼び出する。そして `self.net` は `Sequential` クラスによって変換の列として定義されている。`Sequential` クラスは順伝播の過程を抽象化し、私たちが変換そのものに集中できるようにする。`Sequential` クラスの動作については、Section 9.1.2 でさらに説明する。

学習

学習ループは、ソフトマックス回帰を実装したときとまったく同じである。このモジュール性により、モデルアーキテクチャに関する事項とそれ以外の考慮事項を分離できる。

```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



8.2.3 まとめ

深層ネットワークの設計にさらに慣れてきた今では、深層ネットワークを 1 層から複数層へと拡張することは、それほど大きな難題ではなくなった。特に、学習アルゴリズムとデータローダーは再利用できる。ただし、MLP をゼロから実装するのはそれでも煩雑である。モデルパラメータに名前を付けて管理する必要があるため、モデルの拡張が難しくなる。たとえば、42 層目と 43 層目の間に別の層を挿入したいとししよう。その場合、順番に名前を付け直す覚悟がない限り、それは 42b 層のようなものになってしまうかもしれない。さらに、ネットワークをゼロから実装すると、フレームワークが意味のある性能最適化を行うのははるかに難しくなる。

それでも、完全結合の深層ネットワークがニューラルネットワークモデリングの第一選択だった 1980 年代後半の最先端に、あなたは到達した。次の概念的なステップでは、画像を扱いる。その前に、いくつかの統計の基礎と、モデルを効率よく計算する方法についての詳細を復習する必要がある。

8.2.4 演習

1. 隠れユニット数 `num_hiddens` を変えて、その数がモデルの精度にどう影響するかをプロットして。最適なこのハイパーパラメータの値は何ですか。
2. 隠れ層を追加して、結果にどう影響するか試して。
3. 1 個のニューロンしかない隠れ層を挿入するのが悪い考えなのはなぜですか。何がうまくいかなくなる可能性があるか。
4. 学習率を変えると結果はどう変わりますか。他のすべてのパラメータを固定したとき、どの学習率が最良の結果を与えますか。それはエポック数とどう関係するか。
5. 学習率、エポック数、隠れ層の数、各隠れ層の隠れユニット数を含む、すべてのハイパーパラメータを同時に最適化してみしよう。

    1. それらすべてを最適化したとき、得られる最良の結果は何ですか。
    2. 複数のハイパーパラメータを扱うのがはるかに難しいのはなぜですか。
    3. 複数のパラメータを同時に最適化するための効率的な戦略を説明して。
6. 難しい問題に対して、フレームワーク版とゼロからの実装の速度を比較して。ネットワークの複雑さによってどう変わりますか。
7. 適切に整列した行列と、整列していない行列について、テンソル–行列積の速度を測定して。たとえば、次元が 1024、1025、1026、1028、1032 の行列でテストして。
    1. これは GPU と CPU の間でどう変わりますか。
    2. CPU と GPU のメモリバス幅を求めて。
8. さまざまな活性化関数を試して。どれが最もよく機能するか。
9. ネットワークの重み初期化には違いがあるか。それは重要ですか。

8.3 順伝播、逆伝播、および計算グラフ

これまで、私たちはミニバッチ確率的勾配降下法を用いてモデルを学習してきた。
しかし、アルゴリズムを実装するときに私たちが気にしていたのは、モデルを通した 順伝播に伴う計算だけであった。
勾配を計算するときには、深層学習フレームワークが提供する逆伝播関数を呼び出すだけであった。

勾配の自動計算は、深層学習アルゴリズムの実装を劇的に簡単にする。
自動微分が登場する以前は、複雑なモデルに少し変更を加えるだけでも、複雑な導関数を手計算で再計算する必要があった。
驚くほど頻繁に、学術論文では更新則の導出に何ページも割かなければならなかった。
私たちは引き続き自動微分に頼って興味深い部分に集中する必要があるが、深層学習を表面的にしか理解しないままで終わりたくないのであれば、これらの勾配が内部でどのように計算されているのかを知っておくべきである。

この節では、逆方向伝播（より一般には 逆伝播と呼ばれる）の詳細を深く掘り下げる。
手法とその実装の両方についていくらかの洞察を与えるために、ここでは基本的な数学と計算グラフを用いる。

まずは、重み減衰（ℓ_2 正則化、これは後続の章で説明する）を伴う、1 つの隠れ層を持つ MLP に話を絞る。

8.3.1 順伝播

順伝播（または フォワードパス）とは、入力層から出力層へ向かう順序で、ニューラルネットワークの中間変数（出力を含む）を計算し、保存することを指する。
ここでは、1 つの隠れ層を持つニューラルネットワークの仕組みを、段階を追って見ていくる。
これは退屈に思えるかもしれないが、ファンクの巨匠ジェームス・ブラウンの永遠の言葉を借りれば、“pay the cost to be the boss”である。

簡単のため、入力例が $\mathbf{x} \in \mathbb{R}^d$ であり、隠れ層にバイアス項が含まれないと仮定ししよう。
このとき中間変数は次のようになる。

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \tag{8.3.1}$$

ここで $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ は隠れ層の重みパラメータである。
中間変数 $\mathbf{z} \in \mathbb{R}^h$ を活性化関数 ϕ に通すと、長さ h の隠れ活性化ベクトルが得られる。

$$\mathbf{h} = \phi(\mathbf{z}). \tag{8.3.2}$$

隠れ層の出力 $\mathbf{h}$ もまた中間変数である。
出力層のパラメータが $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ の重みのみを持つと仮定すると、長さ q のベクトルを持つ出力層変数を得られる。

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \tag{8.3.3}$$

損失関数が l であり、データ例のラベルが y であると仮定すると、単一のデータ例に対する損失項は次のように計算できる。

$$L = l(\mathbf{o}, y). \tag{8.3.4}$$

後で導入する ℓ_2 正則化の定義に従い、ハイパーパラメータ λ が与えられたとき、正則化項は

$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_{\mathrm{F}}^2 + \|\mathbf{W}^{(2)}\|_{\mathrm{F}}^2\right), \tag{8.3.5}$$

となる。ここで行列のフロベニウスノルムは、行列をベクトルに平坦化した後に ℓ_2 ノルムを適用したものにすぎない。
最後に、与えられたデータ例に対するモデルの正則化済み損失は次のようになる。

$$J = L + s. \tag{8.3.6}$$

以下では、J を 目的関数と呼ぶ。

8.3.2 順伝播の計算グラフ

計算グラフを描くと、計算内の演算子と変数の依存関係を可視化できる。
Fig. 8.3.1 は、上で述べた単純なネットワークに対応するグラフを示している。
四角は変数、円は演算子を表す。
左下が入力、右上が出力を意味する。
矢印の向き（データの流れを表す）が、主として右方向および上方向であることに注意されたい。

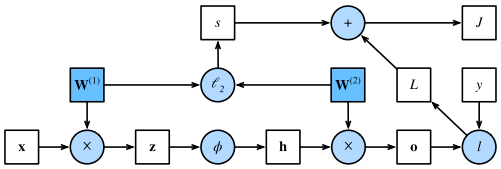


Fig. 8.3.1 順伝播の計算グラフ。

8.3.3 逆伝播

逆伝播とは、ニューラルネットワークのパラメータの勾配を計算する方法を指する。
要するに、この方法は微積分の 連鎖律に従って、ネットワークを出力層から入力層へと逆順にたどる。
このアルゴリズムは、勾配を計算する際に必要となる中間変数（偏導関数）を保存する。
$\mathsf{Y} = f(\mathsf{X})$ および $\mathsf{Z} = g(\mathsf{Y})$ という関数があり、入力と出力 $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ が任意の形状のテンソルであるとする。
連鎖律を用いると、Z の X に関する導関数は次のように計算できる。

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}}\right). \tag{8.3.7}$$

ここでは prod 演算子を用いて、転置や入力位置の入れ替えなど必要な操作を行った後に、その引数を掛け合わせる。
ベクトルの場合、これは単純で、行列同士の積にすぎない。
より高次元のテンソルでは、対応する適切な演算を用いる。
演算子 prod は、記法上の煩雑さをすべて隠している。

Fig. 8.3.1 に計算グラフが示されている、1 つの隠れ層を持つ単純なネットワークのパラメータは、$\mathbf{W}^{(1)}$ と $\mathbf{W}^{(2)}$ でした。
逆伝播の目的は、$\partial J/\partial \mathbf{W}^{(1)}$ と $\partial J/\partial \mathbf{W}^{(2)}$ を計算することである。
これを達成するために、連鎖律を適用し、中間変数とパラメータそれぞれの勾配を順に計算する。
計算の順序は順伝播で行ったものと逆になる。なぜなら、計算グラフの結果から始めて、パラメータへ向かってたどる必要があるからである。
最初のステップは、目的関数 $J = L + s$ の損失項 L と正則化項 s に関する勾配を計算することである。

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1. \tag{8.3.8}$$

次に、連鎖律に従って、目的関数の出力層変数 $\mathbf{o}$ に関する勾配を計算する。

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q. \tag{8.3.9}$$

次に、正則化項の両方のパラメータに関する勾配を計算する。

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}. \tag{8.3.10}$$

これで、出力層に最も近いモデルパラメータの勾配 $\partial J/\partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ を計算できる。
連鎖律を用いると次のようになる。

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}. \tag{8.3.11}$$

$\mathbf{W}^{(1)}$ に関する勾配を得るには、出力層から隠れ層へと逆伝播を続ける必要がある。
隠れ層出力に関する勾配 $\partial J/\partial \mathbf{h} \in \mathbb{R}^h$ は次のように与えられる。

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = \mathbf{W}^{(2)\top} \frac{\partial J}{\partial \mathbf{o}}. \tag{8.3.12}$$

活性化関数 ϕ は要素ごとに適用されるため、中間変数 $\mathbf{z}$ の勾配 $\partial J/\partial \mathbf{z} \in \mathbb{R}^h$ を

計算するには、要素ごとの乗算演算子を用いる必要がある。これを $\odot$ で表する。

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z}). \tag{8.3.13}$$

最後に、入力層に最も近いモデルパラメータの勾配 $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ を得ることができる。
連鎖律に従うと、次を得る。

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}}\mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \tag{8.3.14}$$

8.3.4 ニューラルネットワークの学習

ニューラルネットワークを学習するとき、順伝播と逆伝播は互いに依存している。
特に順伝播では、依存関係の向きに従って計算グラフをたどり、その経路上のすべての変数を計算する。
これらはその後、グラフ上の計算順序が逆になる逆伝播で用いられる。

前述の単純なネットワークを例に考えしよう。
一方で、順伝播中に正則化項 (8.3.5) を計算するには、モデルパラメータ $\mathbf{W}^{(1)}$ と $\mathbf{W}^{(2)}$ の現在値が必要である。
それらは、直近の反復における逆伝播に基づいて最適化アルゴリズムによって与えられる。
他方で、逆伝播中のパラメータ (8.3.11) の勾配計算は、隠れ層出力 $\mathbf{h}$ の現在値に依存しており、それは順伝播によって与えられる。

したがって、ニューラルネットワークを学習するときは、モデルパラメータを初期化した後、順伝播と逆伝播を交互に行い、逆伝播で得られる勾配を用いてモデルパラメータを更新する。
逆伝播は、重複計算を避けるために、順伝播で保存された中間値を再利用することに注意されたい。
その結果の 1 つとして、逆伝播が完了するまで中間値を保持しておく必要がある。
これもまた、学習に平易な予測よりもかなり多くのメモリが必要になる理由の 1 つである。
さらに、そのような中間値のサイズは、おおよそネットワーク層数とバッチサイズに比例する。
したがって、より大きなバッチサイズでより深いネットワークを学習すると、メモリ不足エラーが起こりやすくなる。

8.3.5 まとめ

順伝播は、ニューラルネットワークで定義された計算グラフ内の中間変数を順次計算し、保存する。これは入力層から出力層へ進む。
逆伝播は、ニューラルネットワーク内の中間変数とパラメータの勾配を、逆順に順次計算し、保存する。
深層学習モデルを学習するとき、順伝播と逆伝播は相互依存であり、学習には予測よりもかなり多くのメモリが必要である。

8.3.6 演習

1. あるスカラー関数 f の入力 $\mathbf{X}$ が $n \times m$ 行列であると仮定する。$\mathbf{X}$ に関する f の勾配の次元は何ですか。
2. この節で説明したモデルの隠れ層にバイアスを追加せよ（正則化項にバイアスを含める必要はない）。
    1. 対応する計算グラフを描いて。
    2. 順伝播と逆伝播の式を導出して。
3. この節で説明したモデルにおける学習時と予測時のメモリ使用量を計算して。
4. 2 階導関数を計算したいと仮定する。計算グラフには何が起こりますか。計算にはどれくらい時間がかかると予想するか。
5. 計算グラフが GPU に対して大きすぎると仮定する。
    1. それを複数の GPU に分割できるか。
    2. 小さいミニバッチで学習する場合と比べて、利点と欠点は何ですか。

8.4 数値安定性と初期化

これまでに実装してきたすべてのモデルでは、あらかじめ指定された分布に従ってパラメータを初期化する必要があった。これまでは初期化手法を当然のものとして扱い、その選択がどのように行われるかという詳細には触れてこなかった。そのため、こうした選択はそれほど重要ではないという印象を持ったかもしれない。しかし実際には、初期化手法の選択はニューラルネットワークの学習において重要な役割を果たし、数値安定性を保つうえで決定的になることもある。さらに、これらの選択は非線形活性化関数の選択と興味深い形で結びついている。どの関数を選ぶか、そしてパラメータをどう初期化するかによって、最適化アルゴリズムがどれだけ速く収束するかが決まる。ここでの選択を誤ると、学習中に勾配爆発や勾配消失に遭遇することがある。この節では、こ

れらの話題をより詳しく掘り下げ、深層学習のキャリアを通じて役立ついくつかの有用な経験則を紹介する。

```
%matplotlib inline
from d2l import torch as d2l
import torch
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, np, npx
npx.set_np()
```

```
%matplotlib inline
from d2l import jax as d2l
import jax
from jax import numpy as jnp
from jax import grad, vmap
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf
```

8.4.1 勾配消失と勾配爆発

入力 $\mathbf{x}$ と出力 $\mathbf{o}$ を持つ、L 層の深いネットワークを考える。各層 l は重み $\mathbf{W}^{(l)}$ によってパラメータ化された変換 f_l で定義され、その隠れ層出力を $\mathbf{h}^{(l)}$ とする（$\mathbf{h}^{(0)} = \mathbf{x}$ とする）。このとき、ネットワークは次のように表せる。

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \cdots \circ f_1(\mathbf{x}). \tag{8.4.1}$$

すべての隠れ層出力と入力がベクトルであるとき、$\mathbf{o}$ の任意のパラメータ集合 $\mathbf{W}^{(l)}$ に関する勾配は次のように書ける。

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdots \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}. \tag{8.4.2}$$

言い換えると、この勾配は $L - l$ 個の行列 $\mathbf{M}^{(L)} \cdots \mathbf{M}^{(l+1)}$ と勾配ベクトル $\mathbf{v}^{(l)}$ の積である。したがって、これはあまりにも多くの確率を掛け合わせたときにしばしば現れる数値アンダーフローの問題と同じような影響を受ける。確率を扱うときの一般的な工夫は、対数空間に移ること、すなわち数値表現の仮数部から指数部へと負荷を移すことである。残念ながら、上の問題はそれより深刻である。初期状態では、行列 $\mathbf{M}^{(l)}$ はさまざまな固有値を持ちえる。それらは小さいことも大きいこともあり、その積は 非常に大きくも 非常に小さくもなりえる。

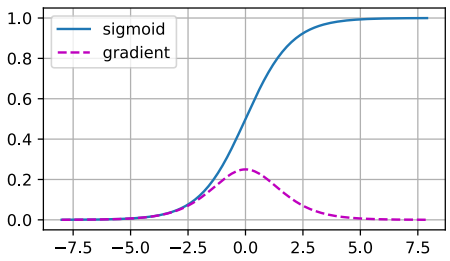
不安定な勾配がもたらす危険は、数値表現の問題にとどまりない。予測不能な大きさの勾配は、最適化アルゴリズムの安定性も脅かする。パラメータ更新が (i) 過度に大きくなってモデルを壊してしまう（勾配爆発問題）か、あるいは (ii) 過度に小さくなって（勾配消失問題）、更新ごとにパラメータがほとんど動かず学習が不可能になるかもしれない。

勾配消失

勾配消失問題のよくある原因の一つは、各層の線形演算の後に付加される活性化関数 σ の選択である。歴史的には、シグモイド関数 $1/(1+\exp(-x))$（Section 8.1 で導入）は、しきい値関数に似ているため人気があった。初期の人工ニューラルネットワークは生物学的ニューラルネットワークに着想を得ていたため、生体ニューロンのように 完全に発火するか まったく発火しないかのどちらかであるニューロンの考え方は魅力的に見えた。シグモイドを詳しく見て、なぜ勾配消失を引き起こしうるのかを確認ししよう。

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.sigmoid(x)
y.backward(torch.ones_like(x))

d2l.plot(x.detach().numpy(), [y.detach().numpy(), x.grad.numpy()],
         legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



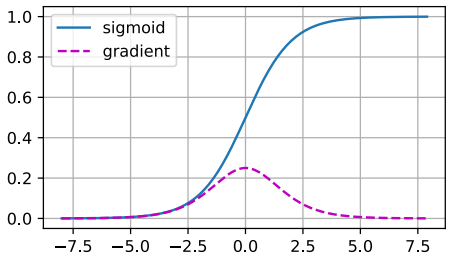

```
x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.sigmoid(x)
y.backward()

d2l.plot(x, [y, x.grad], legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```

```
[07:05:11] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
[07:05:11] ../src/base.cc:48: GPU context requested, but no GPUs found.
```

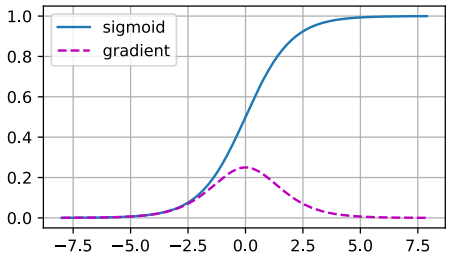
```
x = jnp.arange(-8.0, 8.0, 0.1)
```

(continues on next page)

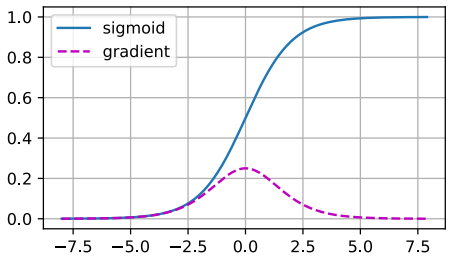


(continued from previous page)

```
y = jax.nn.sigmoid(x)
grad_sigmoid = vmap(grad(jax.nn.sigmoid))
d2l.plot(x, [y, grad_sigmoid(x)],
         legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```




```
x = tf.Variable(tf.range(-8.0, 8.0, 0.1))
with tf.GradientTape() as t:
    y = tf.nn.sigmoid(x)
d2l.plot(x.numpy(), [y.numpy(), t.gradient(y, x).numpy()],
         legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



ご覧のとおり、シグモイドの勾配は、入力が大きいときも小さいときも消失する **）。さらに、多くの層を逆伝播するとき、多くのシグモイドへの入力がゼロに近いちょうどよい範囲にない限り、全体の積の勾配は消失してしまう可能性がある。ネットワークが多層になるほど、注意しないと、どこかの層で勾配が途切れてしまうだろう。実際、この問題はかつて深層ネットワークの学習を悩ませていた。その結果、より安定している（ただし生物学的にはあまりもっ

ともらしくない）ReLU が、実務家にとっての標準的な選択肢として広く使われるようになった。

勾配爆発

逆の問題である勾配爆発も、同様に厄介である。これを少し分かりやすく示すために、100 個のガウス乱数行列を生成し、ある初期行列と掛け合わせてむ。私たちが選んだスケール（分散 $\sigma^2 = 1$ の選択）では、行列積は爆発する。これが深いネットワークの初期化によって起こると、勾配降下法の最適化器を収束させる見込みはありない。

```
M = torch.normal(0, 1, size=(4, 4))
print('a single matrix \n',M)
for i in range(100):
    M = M @ torch.normal(0, 1, size=(4, 4))
print('after multiplying 100 matrices\n', M)
```

```
a single matrix
 tensor([[-0.4847, -0.2991, -0.8154, -0.5333],
        [-0.2325, -1.6432,  0.2363, -0.2251],
        [-0.0026, -0.4851, -0.2048, -0.5896],
        [-0.1705,  0.1220,  0.7720, -0.5366]])
after multiplying 100 matrices
 tensor([[ 1.3606e+24,  3.6219e+24, -2.1369e+24, -3.9221e+23],
        [ 9.3126e+24,  2.4791e+25, -1.4626e+25, -2.6846e+24],
        [ 2.2680e+24,  6.0374e+24, -3.5621e+24, -6.5379e+23],
        [ 1.4751e+24,  3.9267e+24, -2.3167e+24, -4.2521e+23]])
```

```
M = np.random.normal(size=(4, 4))
print('a single matrix', M)
for i in range(100):
    M = np.dot(M, np.random.normal(size=(4, 4)))
print('after multiplying 100 matrices', M)
```

```
a single matrix [[ 2.2122064   1.1630787   0.7740038   0.4838046 ]
 [ 1.0434403   0.29956347  1.1839255   0.15302546]
 [ 1.8917114  -1.1688148  -1.2347414   1.5580711 ]
 [-1.771029   -0.5459446  -0.45138445 -2.3556297 ]]
after multiplying 100 matrices [[ 3.4459747e+23 -7.8040759e+23  5.9973355e+23 ␣
↪4.5230040e+23]
 [ 2.5275059e+23 -5.7240258e+23  4.3988419e+23  3.3174704e+23]
 [ 1.3731275e+24 -3.1097129e+24  2.3897754e+24  1.8022945e+24]
 [-4.4951091e+23  1.0180045e+24 -7.8232368e+23 -5.9000419e+23]]
```

```
get_key = lambda: jax.random.PRNGKey(d2l.get_seed())  # Generate PRNG keys
M = jax.random.normal(get_key(), (4, 4))
print('a single matrix \n', M)
for i in range(100):
```

(continues on next page)

(continued from previous page)

```
    M = jnp.matmul(M, jax.random.normal(get_key(), (4, 4)))
print('after multiplying 100 matrices\n', M)
```

```
a single matrix
 [[-0.01468026  0.18429002  0.29529375  1.3654956 ]
 [-0.3123339   0.15577619 -1.126514   -0.279658  ]
 [-0.17748299  1.1886835   0.24319468 -0.02573   ]
 [ 0.8505334   0.4623113  -1.6157581   0.48986113]]
after multiplying 100 matrices
 [[ 1.5799222e+22 -3.6156512e+23 -5.8382020e+21 -1.8848757e+23]
 [ 3.1915137e+22 -7.3038060e+23 -1.1793621e+22 -3.8075463e+23]
 [-1.8222258e+22  4.1701538e+23  6.7335221e+21  2.1739437e+23]
 [-9.0227682e+21  2.0648444e+23  3.3340368e+21  1.0764252e+23]]
```

```
M = tf.random.normal((4, 4))
print('a single matrix \n', M)
for i in range(100):
    M = tf.matmul(M, tf.random.normal((4, 4)))
print('after multiplying 100 matrices\n', M.numpy())
```

```
a single matrix
 tf.Tensor(
[[ 1.1841711   0.6504623  -0.29817837  0.37453428]
 [-1.0665823  -0.09414102 -2.4775772   0.5071953 ]
 [-1.0149585   0.07600389  2.1217499   0.8125305 ]
 [ 1.6692833  -0.18011202 -1.1919788  -0.01561711]], shape=(4, 4),␣
↪dtype=float32)
after multiplying 100 matrices
 [[ 3.6784546e+24 -5.8023196e+23  2.4111507e+24  1.9948421e+24]
 [-7.0356487e+24  1.1097884e+24 -4.6117209e+24 -3.8154634e+24]
 [-2.0623473e+24  3.2531121e+23 -1.3518255e+24 -1.1184203e+24]
 [ 4.4600454e+24 -7.0351904e+23  2.9234677e+24  2.4187031e+24]]
```

対称性を破る

ニューラルネットワーク設計におけるもう一つの問題は、パラメータ化に内在する対称性である。1 つの隠れ層と 2 つのユニットを持つ単純な MLP を考える。この場合、最初の層の重み $\mathbf{W}^{(1)}$ を入れ替え、同様に出力層の重みも入れ替えることで、同じ関数を得ることができる。第 1 隠れユニットと第 2 隠れユニットを区別する特別なものはありない。言い換えると、各層の隠れユニットの間には置換対称性がある。

これは単なる理論上の厄介ごとではありない。先ほどの 1 隠れ層・2 ユニットの MLP を考えしよう。説明のために、出力層が 2 つの隠れユニットを 1 つの出力ユニットに変換するとする。もし隠れ層のすべてのパラメータを $\mathbf{W}^{(1)} = c$（ある定数 c）として初期化したら、何が起こるだろうか。この場合、順伝播ではどちらの隠れユニットも同じ入力とパラメータを受け取り、同じ活性化を生成し、それが出力ユニットに渡される。逆伝播では、出力ユニットをパラメー

タ $\mathbf{W}^{(1)}$ で微分すると、すべての要素が同じ値を取る勾配が得られる。したがって、勾配ベースの反復（たとえばミニバッチ確率的勾配降下法）を行っても、$\mathbf{W}^{(1)}$ のすべての要素は依然として同じ値のままである。このような反復だけでは自力で対称性を 破ることは決してできず、ネットワークの表現力を実現できないままになるかもしれない。隠れ層は、あたかも 1 つのユニットしか持たないかのように振る舞うだろう。ミニバッチ確率的勾配降下法ではこの対称性は破れないが、（後で導入する）ドロップアウト正則化なら破ることができる点に注意されたい。

8.4.2 パラメータ初期化

上で述べた問題に対処する、あるいは少なくとも軽減する一つの方法は、慎重な初期化である。後で見るように、最適化時にさらに注意を払い、適切な正則化を行うことで、安定性をさらに高められる。

デフォルトの初期化

前の節、たとえば Section 6.5 では、重みの値を初期化するために正規分布を用いた。初期化方法を指定しない場合、フレームワークはデフォルトのランダム初期化方法を使いる。これは中程度の問題規模では実際によく機能することが多いである。

Xavier 初期化

非線形性 なしのある全結合層について、出力 o_i のスケール分布を見てみしよう。この層に n_{in} 個の入力 x_j と、それに対応する重み w_{ij} があるとすると、出力は次のように与えられる。

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j. \tag{8.4.3}$$

重み w_{ij} はすべて同じ分布から独立にサンプルされるとする。さらに、この分布の平均が 0、分散が σ^2 であると仮定する。ここで、分布がガウス分布である必要はなく、平均と分散が存在すればよいことに注意されたい。今のところ、この層への入力 x_j も平均 0、分散 γ^2 を持ち、w_{ij} と互いに独立で、かつ入力同士も独立であると仮定ししよう。この場合、o_i の平均は次のように計算できる。

$$\begin{aligned} E[o_i] &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j] \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}] E[x_j] \\ &= 0, \end{aligned} \tag{8.4.4}$$

また分散は次のようになる。

$$\begin{aligned}\mathrm{Var}[o_i] &= E[o_i^2] - (E[o_i])^2 \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0 \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2] E[x_j^2] \\ &= n_{\text{in}} \sigma^2 \gamma^2.\end{aligned} \tag{8.4.5}$$

分散を一定に保つ一つの方法は、$n_{\text{in}}\sigma^2 = 1$ とすることである。次に逆伝播を考える。ここでも同様の問題に直面するが、今度は勾配が出力に近い層から伝播してくる。順伝播の場合と同じ考え方を使うと、$n_{\text{out}}\sigma^2 = 1$ でない限り、勾配の分散は爆発しうることが分かる。ここで n_{out} はこの層の出力数である。すると、私たちはジレンマに陥る。この 2 つの条件を同時に満たすことはできない。そこで、次を満たすことを目指す。

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}. \tag{8.4.6}$$

これが、現在では標準的で実用上も有益な *Xavier* 初期化の理論的根拠である。この手法は、その考案者の第一著者にちなんで名付けられた (Glorot and Bengio, 2010)。通常、Xavier 初期化では平均 0、分散 $\sigma^2 = \frac{2}{n_{\text{in}}+n_{\text{out}}}$ のガウス分布から重みをサンプルする。また、重みを一様分布からサンプルするときの分散の選び方にも適用できる。一様分布 $U(-a, a)$ の分散は $\frac{a^2}{3}$ であることに注意されたい。$\frac{a^2}{3}$ を σ^2 に関する条件へ代入すると、次のように初期化すればよいことが分かる。

$$U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right). \tag{8.4.7}$$

上の数学的な議論では非線形性がないことを仮定しているが、この仮定はニューラルネットワークでは簡単に破られる。それでも、Xavier 初期化法は実際にはうまく機能することが分かっている。

さらに先へ

上の議論は、現代的なパラメータ初期化手法のほんの入口にすぎない。深層学習フレームワークには、しばしば十数種類もの異なる経験則が実装されている。さらに、パラメータ初期化は深層学習における基礎研究の非常に活発な分野であり続けている。そこには、共有パラメータ、超解像、系列モデル、その他の状況に特化した経験則も含まれる。たとえば、Xiao *et al.* (2018) は、慎重に設計された初期化法を用いることで、アーキテクチャ上の工夫なしに 10,000 層のニューラルネットワークを学習できる可能性を示した。

この話題に興味があるなら、このモジュールで扱う各手法を深く掘り下げ、それぞれの経験則を提案・解析した論文を読み、さらにこの分野の最新論文を追ってみることを勧める。もしかすると、あなたは巧妙なアイデアを見つけた

り、あるいは発明したりして、深層学習フレームワークに実装を貢献するかもしれない。

8.4.3 まとめ

勾配消失と勾配爆発は、深いネットワークでよく見られる問題である。勾配とパラメータが適切に制御された状態を保つためには、パラメータ初期化に細心の注意が必要である。初期勾配が大きすぎず小さすぎもしないようにするために、初期化の経験則が必要である。ランダム初期化は、最適化の前に対称性が破られることを保証するうえで重要である。Xavier 初期化は、各層について、任意の出力の分散が入力数の影響を受けず、任意の勾配の分散が出力数の影響を受けないことを示唆する。ReLU 活性化関数は勾配消失問題を緩和する。これにより収束を加速できる。

8.4.4 演習

1. MLP の各層における置換対称性以外に、対称性を破る必要があるニューラルネットワークのケースを他に設計できるか？
2. 線形回帰や softmax 回帰では、すべての重みパラメータを同じ値に初期化してもよいだろうか？
3. 2 つの行列の積の固有値に関する解析的な上界を調べて。これは、勾配がよく条件付けられるようにすることについて何を示唆しているか？
4. ある項が発散すると分かっている場合、後から修正できるだろうか？layer-wise adaptive rate scaling に関する論文を参考にされたい (You *et al.*, 2017)。

8.5 ディープラーニングにおける汎化

Section 6 と Section 7 では、線形モデルを訓練データに当てはめることで回帰問題と分類問題に取り組んだ。どちらの場合も、観測された訓練ラベルの尤度を最大化するパラメータを見つけるための実用的なアルゴリズムを示した。そして各章の終わり近くで、訓練データへの当てはめはあくまで中間目標にすぎないことを思い出した。私たちの本当の目的は一貫して、同じ背後の母集団から引き出された新しいデータ例に対しても正確な予測を行えるような 一般的なパターンを見つけることであった。機械学習研究者は最適化アルゴリズムの 消費者である。時には、新しい最適化アルゴリズムを開発しなければならないことさえある。しかし結局のところ、最適化は目的を達成するための手段にすぎない。本質的には、機械学習は統計学の一分野であり、訓練損失を最適化したいのは、何らかの統計的原理（既知であれ未知であれ）によって得られたモデルが訓練集合を超えて汎化する場合に限られる。

明るい側面としては、確率的勾配降下法で訓練された深層ニューラルネット

ワークは、コンピュータビジョン、自然言語処理、時系列データ、推薦システム、電子健康記録、タンパク質折りたたみ、ビデオゲームやボードゲームにおける価値関数近似、そしてその他数多くの領域にまたがる無数の予測問題で驚くほどよく汎化することが分かっている。一方で、最適化の物語（なぜ訓練データに当てはめられるのか）あるいは汎化の物語（なぜ得られたモデルが未見のデータ例に汎化するのか）のどちらかについて分かりやすい説明を求めているなら、一杯飲みたくなるかもしれない。線形モデルの最適化手順と解の統計的性質については、包括的な理論体系によって十分よく記述されているが、深層学習についての理解は、両面ともいまだに無法地帯の様相を呈している。

深層学習の理論と実践の両方は急速に進化しており、理論家たちは何が起きているのかを説明するために新しい戦略を採用し続けている。その一方で実践者たちも、実践的なヒューリスティクス手法を蓄積し、どの技法をどの状況で適用すべきかを判断するための直感や経験則の知識体系を、猛烈な速度で革新し続けている。

現時点での要約は、深層学習の理論は有望な攻め口と散発的に興味深い結果を生み出してはいるものの、(i) なぜニューラルネットワークを最適化できるのか、および (ii) 勾配降下法で学習されたモデルが高次元の課題でさえもなぜこれほどよく汎化するのか、という両方について包括的に説明するにはまだほど遠いように見える、ということである。しかし実践上は、(i) はめったに問題になりない（訓練データすべてに当てはまるパラメータは常に見つけられる）ので、汎化を理解することのほうがはるかに大きな問題である。一方で、一貫した科学理論の安心感がなくても、実践者たちは実際にうまく汎化するモデルを作るのに役立つ多くの技法を開発してきた。深層学習における汎化という広大な話題を簡潔な要約だけで十分に語り尽くすことはできないし、研究全体の状況もまだ決着にはほど遠いであるが、本節では研究と実践の現状を広く概観したいと思いる。

8.5.1 過学習と正則化の再考

Wolpert and Macready (1995) の「ノーフリーランチ」定理によれば、どの学習アルゴリズムも、ある分布のデータではよりよく汎化し、別の分布のデータではより悪く汎化する。したがって、有限の訓練集合が与えられたとき、モデルはある種の仮定に依存する。人間並みの性能を達成するには、人間が世界をどう考えるかを反映した 帰納バイアスを特定することが有用かもしれない。このような帰納バイアスは、ある性質をもつ解に対する選好を与える。たとえば、深い MLP は、より単純な関数の合成によって複雑な関数を構成する方向への帰納バイアスを持っている。

機械学習モデルが帰納バイアスをエンコードしている以上、それらを訓練する私たちのアプローチは通常、(i) 訓練データに当てはめること、および (ii) ホールドアウトデータ上でモデルを評価することによって 汎化誤差（背後の母集団における真の誤差）を推定すること、という 2 段階から成る。訓練データへの当てはまりとテストデータへの当てはまりの差は 汎化ギャップと呼ばれ、これが大きいとき、モデルは訓練データに 過学習していると言いる。過学習が極端な場

合には、テスト誤差が依然として大きいにもかかわらず、訓練データに完全に一致させてしまうことがある。古典的な見方では、これはモデルが複雑すぎることを意味し、特徴量の数、学習される非ゼロパラメータの数、あるいはパラメータの大きさを何らかの尺度で縮小する必要があると解釈される。モデルの複雑さと損失の関係を示した図（Fig. 6.6.1）を Section 6.6 で思い出して。

しかし深層学習は、この図式を直感に反する形で複雑にする。まず、分類問題では、私たちのモデルは通常、数百万個からなるデータセットであってもすべての訓練データ例を完全に当てはめられるほど表現力がある (Zhang *et al.*, 2021)。古典的な図式では、この状況はモデル複雑度軸のはるか右端に位置し、汎化誤差の改善はすべて正則化によってもたらされるはずだと考えるかもしれない。つまり、モデルクラスの複雑さを下げるか、あるいはペナルティを課して、パラメータが取りうる値の集合を厳しく制約するかである。しかし、そこから話が奇妙になり始める。

不思議なことに、多くの深層学習タスク（たとえば画像認識やテキスト分類）では、私たちは通常、複数のモデルアーキテクチャの中から選んでおり、それらはいずれも訓練損失を任意に小さく（そして訓練誤差をゼロに）できる。検討中のすべてのモデルが訓練誤差ゼロを達成する以上、さらなる改善の唯一の道は過学習を減らすことである。さらに奇妙なことに、訓練データに完全に当てはめたにもかかわらず、モデルを さらに表現力豊かにするたとえば層やノードを追加する、あるいはより多くのエポックで訓練することで、実際には汎化誤差をさらに減らせることがよくある。さらに奇妙なのは、モデルの 複雑さ（たとえばネットワークの深さや幅で捉えられるもの）と汎化ギャップの関係が単調でないことがあり、複雑さが最初は悪化を招くものの、その後いわゆる「ダブルディセント」パターンで改善に転じることがある (Nakkiran *et al.*, 2021)。このように深層学習の実践者は、一部はモデルを何らかの形で制約しているように見え、別の一部は逆にモデルをより表現力豊かにしているように見えるさまざまな技法を使い分けており、それらはすべて、ある意味では過学習を抑えるために適用されている。

さらに事態を複雑にしているのは、古典的学習理論が与える保証は古典的モデルに対してでさえ保守的でありうる一方で、そもそもなぜ深層ニューラルネットワークが汎化するのかを説明する力はないように見えることである。深層ニューラルネットワークは、大規模データセットであっても任意のラベルを当てはめることができ、ℓ_2 正則化のような馴染み深い手法を使っていても、VC 次元や仮説クラスの Rademacher 複雑度に基づく従来の複雑度ベースの汎化境界は、ニューラルネットワークがなぜ汎化するのかを説明できない。

8.5.2 非パラメトリック手法からの着想

初めて深層学習に触れると、それらをパラメトリックモデルだと考えたくなる。実際、モデルには何百万ものパラメータがある。モデルを更新するとき、私たちはそのパラメータを更新する。モデルを保存するとき、私たちはそのパラメータをディスクに書き出する。しかし、数学と計算機科学には、直感に反する視点の転換や、一見異なる問題の間にある驚くべき同型性に満ちている。ニューラルネットワークは確かにパラメータを 持っているが、ある意味では

非パラメトリックモデルのように振る舞うと考えるほうがより有益なことがある。では、何がモデルを非パラメトリックにするのだろうか。この名称は多様なアプローチを含むが、共通するテーマの一つは、非パラメトリック手法は利用可能なデータ量が増えるにつれて複雑さのレベルも増していく傾向があることである。

おそらく最も単純な非パラメトリックモデルの例は、k 近傍法です（より多くの非パラメトリックモデルについては、たとえば Section 14.2 で後ほど扱いる）。ここでは、訓練時に学習器は単にデータセットを記憶する。そして予測時に、新しい点 $\mathbf{x}$ に直面すると、学習器は k 個の最近傍（ある距離 $d(\mathbf{x}, \mathbf{x}'_i)$ を最小化する k 個の点 $\mathbf{x}'_i$）を探する。$k = 1$ のとき、このアルゴリズムは 1-最近傍法と呼ばれ、このアルゴリズムは常に訓練誤差ゼロを達成する。しかし、それはこのアルゴリズムが汎化しないことを意味しない。実際、いくつかの穏やかな条件の下では、1-最近傍アルゴリズムは整合的である（最終的に最適予測器へ収束する）ことが分かっている。

1-最近傍法では、何らかの距離関数 d を指定する必要があること、あるいは同値に、データを特徴量化するためのベクトル値基底関数 $\phi(\mathbf{x})$ を指定する必要があることに注意されたい。どの距離尺度を選んでも、訓練誤差ゼロを達成し、やがて最適予測器に到達するが、距離尺度 d が異なれば異なる帰納バイアスをエンコードし、有限のデータ量では異なる予測器を与える。距離尺度 d の選択の違いは、背後にあるパターンについての異なる仮定を表しており、異なる予測器の性能は、それらの仮定が観測データとどれだけ整合的かに依存する。

ある意味で、ニューラルネットワークは過剰パラメータ化されており、訓練データに当てはめるのに必要な数よりもはるかに多くのパラメータを持っているため、訓練データを 補間する（完全に当てはめる）傾向があり、そのためいくつかの点では非パラメトリックモデルにより近い振る舞いをする。より最近の理論研究は、大規模ニューラルネットワークと非パラメトリック手法、特にカーネル法との間に深い関係があることを明らかにした。とりわけ Jacot *et al.* (2018) は、ランダム初期化された重みを持つ多層パーセプトロンが無限に幅広くなる極限で、特定のカーネル関数（本質的には距離関数）の選び方に対して（非パラメトリックな）カーネル法と等価になることを示した。彼らはこのカーネルをニューラル接線カーネルと呼んでいる。現在のニューラル接線カーネルモデルは現代の深層ネットワークの振る舞いを完全には説明できないかもしれないが、解析ツールとしての成功は、過剰パラメータ化された深層ネットワークの振る舞いを理解するうえで非パラメトリックモデリングが有用であることを強く示している。

8.5.3 早期停止

深層ニューラルネットワークは、ラベルが誤って付与された場合やランダムに付与された場合でさえ任意のラベルを当てはめることができるが (Zhang *et al.*, 2021)、この能力は訓練の多くの反復を経て初めて現れる。ラベルノイズの設定における新しい研究 (Rolnick *et al.*, 2017) は、ニューラルネットワークはまず正しくラベル付けされたデータを当てはめ、その後になって初めて誤ラベルのデー

タを補間する傾向があることを明らかにした。さらに、この現象が汎化に関する保証へ直接つながることも示されている。すなわち、モデルが正しくラベル付けされたデータには当てはめているが、訓練集合に含まれるランダムにラベル付けされたデータ例にはまだ当てはめていないなら、その時点で実際には汎化しているのである (Garg *et al.*, 2021)。

これらの知見は、深層ニューラルネットワークの正則化における古典的な手法である 早期停止を動機づける。ここでは、重みの値を直接制約する代わりに、訓練エポック数を制約する。停止基準を決める最も一般的な方法は、訓練中ずっと検証誤差を監視し（通常は各エポックの後に 1 回確認し）、検証誤差がある小さな量 ϵ 以上減少しなくなってから一定数のエポックが経過した時点で訓練を打ち切ることである。これは 忍耐基準と呼ばれることがある。ノイズのあるラベル設定でよりよい汎化につながる可能性があることに加えて、早期停止のもう一つの利点は時間の節約である。いったん忍耐基準を満たせば、訓練を終了できる。8 個以上の GPU 上で同時に数日間の訓練を要するような大規模モデルでは、うまく調整された早期停止によって研究者の時間を何日も節約でき、雇用主にとっては何千ドルものコスト削減になる。

特に、ラベルノイズがなく、データセットが 実現可能である（クラスが本当に分離可能である、たとえば猫と犬を区別する場合）とき、早期停止は汎化の大きな改善にはつながりにくい傾向がある。一方で、ラベルノイズがある場合や、ラベルに本質的な変動がある場合（たとえば患者の死亡率予測）には、早期停止が不可欠である。ノイズのあるデータを補間するまでモデルを訓練し続けるのは、通常は悪い考えである。

8.5.4 深層ネットワークのための古典的正則化手法

Section 6 では、モデルの複雑さを制約するためのいくつかの古典的な正則化技法を説明した。特に Section 6.7 では、重み減衰と呼ばれる手法を導入した。これは、大きな重みの値にペナルティを課すために損失関数に正則化項を加えるものである。どの重みノルムにペナルティを課すかによって、この技法はリッジ正則化（ℓ_2 ペナルティの場合）またはラッソ正則化（ℓ_1 ペナルティの場合）として知られる。これらの正則化の古典的な解析では、モデルが任意のラベルを当てはめるのを防ぐのに十分なほど重みの取りうる値を制限すると考えられている。

深層学習の実装では、重み減衰はいまなお人気のある手法である。しかし研究者たちは、典型的な強さの ℓ_2 正則化ではネットワークがデータを補間するのを防ぐには不十分であることを指摘しており (Zhang *et al.*, 2021)、したがって正則化として解釈した場合の利点は、早期停止基準と組み合わせたときにのみ意味を持つのかもしれない。早期停止がなければ、深層学習における層数やノード数、あるいは 1-最近傍法における距離尺度と同様に、これらの手法はニューラルネットワークの能力を実質的に制約するからではなく、むしろデータセットに見られるパターンとより相性のよい帰納バイアスを何らかの形でエンコードしているからこそ、よりよい汎化をもたらすのかもしれない。したがって、古典的な正則化手法は、その有効性の理論的根拠が大きく異なっている可能性があっても、深層学習の実装では依然として広く使われている。

特筆すべきことに、深層学習研究者たちは、モデル入力にノイズを加えるといった古典的な正則化の文脈で最初に普及した技法も発展させてきた。次節では、有名な dropout 技法（Srivastava *et al.* (2014) によって発明された）を紹介する。これは深層学習の定番となっているが、その有効性の理論的基盤は同様に謎のままである。

8.5.5 まとめ

古典的な線形モデルとは異なり、深層ネットワークは通常、データ例の数よりも少ないパラメータではなく、過剰パラメータ化されており、ほとんどのタスクで訓練集合を完全に当てはめることができる。この 補間領域は、多くの強く持たれてきた直感に挑戦する。機能的には、ニューラルネットワークはパラメトリックモデルのように見える。しかし、非パラメトリックモデルとして考えるほうが、直感のより信頼できる源になることがある。検討中のすべての深層ネットワークが訓練ラベルすべてを当てはめられることが多いため、ほとんどすべての改善は過学習を抑えること（汎化ギャップを埋めること）によってもたらされなければなりない。逆説的であるが、汎化ギャップを減らす介入は、あるときにはモデルの複雑さを増しているように見え、別のときには複雑さを減らしているように見える。しかし、これらの手法は古典理論が深層ネットワークの汎化を説明できるほどには複雑さを十分に下げることはめったになく、なぜ特定の選択が汎化の改善につながるのかは、多くの優れた研究者たちの集中的な努力にもかかわらず、大部分が巨大な未解決問題のままである。

8.5.6 演習

1. 従来の複雑度ベースの尺度は、深層ニューラルネットワークの汎化をどのような意味で説明できないのだろうか。
2. 早期停止はなぜ正則化手法と見なせるのだろうか。
3. 研究者は通常、停止基準をどのように決めるのだろうか。
4. 早期停止が汎化を大きく改善する場合を分ける重要な要因は何だと考えられますか。
5. 汎化以外に、早期停止のもう一つの利点を述べて。

8.6 ドロップアウト

まず、優れた予測モデルに何を期待するかを簡単に考えてみしよう。私たちは、未知のデータに対して良い性能を発揮してほしいと考える。古典的な汎化理論は、訓練性能とテスト性能の差を縮めるには、単純なモデルを目指すべきだと示唆している。単純さは、次元数が少ないという形で現れる。これは、Section 6.6 で線形モデルの単項式基底関数を議論したときに見た。さらに、

Section 6.7 で重み減衰（ℓ_2 正則化）を議論したときに見たように、パラメータの（逆）ノルムも単純さの有用な尺度を表する。単純さのもう 1 つの有用な概念は滑らかさ、すなわち、関数が入力の小さな変化に敏感でないことである。たとえば、画像を分類するときには、画素に多少のランダムノイズを加えてもほとんど問題ないはずだと期待する。

Bishop (1995) は、入力ノイズを加えて学習することがチホノフ正則化と等価であることを証明し、この考えを形式化した。この研究は、関数が滑らかである（したがって単純である）ことの要請と、入力の摂動に対して頑健であることの要請との間に、明確な数学的つながりを与えた。

その後、Srivastava *et al.* (2014) は、Bishop の考えをネットワークの内部層にも適用する巧妙なアイデアを開発した。ドロップアウトと呼ばれるこのアイデアは、順伝播の各内部層を計算する際にノイズを注入するもので、ニューラルネットワークの学習における標準的な手法となっている。この手法が *dropout* と呼ばれるのは、学習中に実際にいくつかのニューロンを *drop out*（脱落）させるからである。学習の各反復で、標準的なドロップアウトは、次の層を計算する前に各層のノードの一定割合をゼロにする。

明確にしておくと、Bishop とのつながりは私たちが独自に付けた説明である。ドロップアウトの元論文は、性生殖との驚くべき類推を通じて直感を与えている。著者らは、ニューラルネットワークの過学習は、各層が前の層の特定の活性化パターンに依存している状態として特徴づけられると主張し、この状態を *co-adaptation* と呼んでいる。彼らによれば、ドロップアウトは co-adaptation を壊し、性生殖が co-adapted な遺伝子を壊すとされるのと同じだといいる。この理論の正当化は確かに議論の余地があるが、ドロップアウト手法そのものは長く使われ続けており、さまざまな形式のドロップアウトがほとんどの深層学習ライブラリに実装されている。

重要な課題は、このノイズをどのように注入するかである。1 つの考え方は、各層の期待値が—他の層を固定したときに—ノイズがない場合に取る値と等しくなるように、不偏な方法で注入することである。Bishop の研究では、線形モデルの入力にガウスノイズを加えた。各学習反復で、平均 0 の分布からサンプルしたノイズ $\epsilon \sim \mathcal{N}(0, \sigma^2)$ を入力 $\mathbf{x}$ に加え、摂動された点 $\mathbf{x}' = \mathbf{x} + \epsilon$ を得た。期待値では、$E[\mathbf{x}'] = \mathbf{x}$ である。

標準的なドロップアウト正則化では、各層のノードの一定割合をゼロにし、その後、保持された（ドロップアウトされなかった）ノードの割合で正規化することで、各層を 補正する。言い換えると、ドロップアウト確率 p に対して、各中間活性化 h は次のようにランダム変数 h' に置き換えられる。

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases} \tag{8.6.1}$$

設計上、期待値は変わらず、すなわち $E[h'] = h$ である。

```
from d2l import torch as d2l
```

(continues on next page)

(continued from previous page)

```
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from functools import partial
import jax
from jax import numpy as jnp
import optax
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

8.6.1 実践におけるドロップアウト

Fig. 8.1.1 の、隠れ層 1 つと 5 個の隠れユニットを持つ MLP を思い出してほしい。隠れ層にドロップアウトを適用し、各隠れユニットを確率 p でゼロにすると、その結果は元のニューロンの部分集合だけを含むネットワークと見なせる。Fig. 8.6.1 では、h_2 と h_5 が取り除かれている。その結果、出力の計算はもはや h_2 や h_5 に依存せず、逆伝播を行うときにはそれぞれの勾配も消える。このようにして、出力層の計算が $h_1, \ldots, h_5$ のどれか 1 つの要素に過度に依存することを防げる。

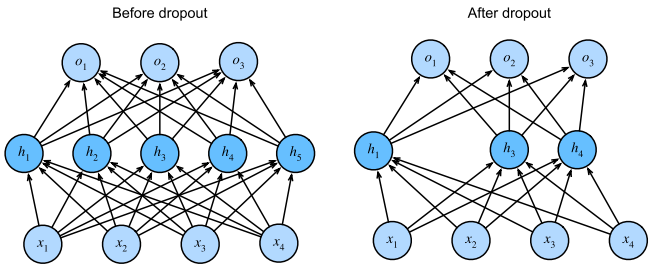


Fig. 8.6.1 ドロップアウト前後の MLP。

通常、テスト時にはドロップアウトを無効にする。学習済みモデルと新しいデータ例が与えられたときには、どのノードもドロップアウトしないため、正規化も不要である。ただし、例外もある。一部の研究者は、ニューラルネットワ

ークの予測の 不確実性を推定するヒューリスティックとして、テスト時にもドロップアウトを使いる。もし多くの異なるドロップアウト出力で予測が一致するなら、そのネットワークはより確信を持っていると言えるかもしれない。

8.6.2 ゼロからの実装

単一層のドロップアウト関数を実装するには、層の次元数と同じだけのベルヌーイ（二値）確率変数からサンプルを生成する必要がある。この確率変数は、確率 $1-p$ で値 1（保持）を、確率 p で値 0（ドロップ）を取る。これを実装する簡単な方法は、まず一様分布 $U[0,1]$ からサンプルを生成することである。そして、対応するサンプルが p より大きいノードを保持し、それ以外をドロップする。

以下のコードでは、テンソル入力 `X` の要素を確率 `dropout` でドロップする `dropout_layer` 関数を実装し、上で説明したように残りを再スケーリングする。つまり、生き残った要素を `1.0-dropout` で割る。

```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    if dropout == 1: return torch.zeros_like(X)
    mask = (torch.rand(X.shape) > dropout).float()
    return mask * X / (1.0 - dropout)
```

```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    if dropout == 1: return np.zeros_like(X)
    mask = np.random.uniform(0, 1, X.shape) > dropout
    return mask.astype(np.float32) * X / (1.0 - dropout)
```

```
def dropout_layer(X, dropout, key=d2l.get_key()):
    assert 0 <= dropout <= 1
    if dropout == 1: return jnp.zeros_like(X)
    mask = jax.random.uniform(key, X.shape) > dropout
    return jnp.asarray(mask, dtype=jnp.float32) * X / (1.0 - dropout)
```

```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    if dropout == 1: return tf.zeros_like(X)
    mask = tf.random.uniform(
        shape=tf.shape(X), minval=0, maxval=1) < 1 - dropout
    return tf.cast(mask, dtype=tf.float32) * X / (1.0 - dropout)
```

いくつかの例で `dropout_layer` 関数を試してみましょう。以下のコードでは、入力 `X` をそれぞれ確率 0、0.5、1 のドロップアウト操作に通する。

```
if tab.selected('mxnet'):
```

(continues on next page)

(continued from previous page)

```
    X = np.arange(16).reshape(2, 8)
if tab.selected('pytorch'):
    X = torch.arange(16, dtype = torch.float32).reshape((2, 8))
if tab.selected('tensorflow'):
    X = tf.reshape(tf.range(16, dtype=tf.float32), (2, 8))
if tab.selected('jax'):
    X = jnp.arange(16, dtype=jnp.float32).reshape(2, 8)
print('dropout_p = 0:', dropout_layer(X, 0))
print('dropout_p = 0.5:', dropout_layer(X, 0.5))
print('dropout_p = 1:', dropout_layer(X, 1))
```

```
dropout_p = 0: tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
dropout_p = 0.5: tensor([[ 0.,  2.,  0.,  0.,  0., 10., 12., 14.],
        [16.,  0., 20.,  0., 24.,  0.,  0.,  0.]])
dropout_p = 1: tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
```

モデルの定義

以下のモデルは、各隠れ層の出力（活性化関数の後）にドロップアウトを適用する。層ごとに別々のドロップアウト確率を設定できる。一般的には、入力層に近いほどドロップアウト確率を低く設定する。ドロップアウトは学習時にのみ有効になるようにする。

```
class DropoutMLPScratch(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr):
        super().__init__()
        self.save_hyperparameters()
        self.lin1 = nn.LazyLinear(num_hiddens_1)
        self.lin2 = nn.LazyLinear(num_hiddens_2)
        self.lin3 = nn.LazyLinear(num_outputs)
        self.relu = nn.ReLU()

    def forward(self, X):
        H1 = self.relu(self.lin1(X.reshape((X.shape[0], -1))))
        if self.training:
            H1 = dropout_layer(H1, self.dropout_1)
        H2 = self.relu(self.lin2(H1))
        if self.training:
            H2 = dropout_layer(H2, self.dropout_2)
        return self.lin3(H2)
```

```
class DropoutMLPScratch(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr):
        super().__init__()
        self.save_hyperparameters()
```

(continues on next page)

(continued from previous page)

```
        self.lin1 = nn.Dense(num_hiddens_1, activation='relu')
        self.lin2 = nn.Dense(num_hiddens_2, activation='relu')
        self.lin3 = nn.Dense(num_outputs)
        self.initialize()

    def forward(self, X):
        H1 = self.lin1(X)
        if autograd.is_training():
            H1 = dropout_layer(H1, self.dropout_1)
        H2 = self.lin2(H1)
        if autograd.is_training():
            H2 = dropout_layer(H2, self.dropout_2)
        return self.lin3(H2)
```

```
class DropoutMLPScratch(d2l.Classifier):
    num_hiddens_1: int
    num_hiddens_2: int
    num_outputs: int
    dropout_1: float
    dropout_2: float
    lr: float
    training: bool = True

    def setup(self):
        self.lin1 = nn.Dense(self.num_hiddens_1)
        self.lin2 = nn.Dense(self.num_hiddens_2)
        self.lin3 = nn.Dense(self.num_outputs)
        self.relu = nn.relu

    def forward(self, X):
        H1 = self.relu(self.lin1(X.reshape(X.shape[0], -1)))
        if self.training:
            H1 = dropout_layer(H1, self.dropout_1)
        H2 = self.relu(self.lin2(H1))
        if self.training:
            H2 = dropout_layer(H2, self.dropout_2)
        return self.lin3(H2)
```

```
class DropoutMLPScratch(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr):
        super().__init__()
        self.save_hyperparameters()
        self.lin1 = tf.keras.layers.Dense(num_hiddens_1, activation='relu')
        self.lin2 = tf.keras.layers.Dense(num_hiddens_2, activation='relu')
        self.lin3 = tf.keras.layers.Dense(num_outputs)

    def forward(self, X):
        H1 = self.lin1(tf.reshape(X, (X.shape[0], -1)))
        if self.training:
            H1 = dropout_layer(H1, self.dropout_1)
        H2 = self.lin2(H1)
        if self.training:
```

(continues on next page)

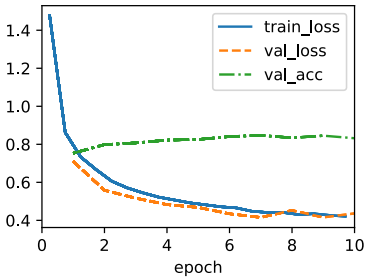
(continued from previous page)

```
            H2 = dropout_layer(H2, self.dropout_2)
        return self.lin3(H2)
```

学習

以下は、先に説明した MLP の学習と同様である。

```
hparams = {'num_outputs':10, 'num_hiddens_1':256, 'num_hiddens_2':256,
           'dropout_1':0.5, 'dropout_2':0.5, 'lr':0.1}
model = DropoutMLPScratch(**hparams)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



8.6.3 簡潔な実装

高水準 API を使う場合、各全結合層の後に Dropout 層を追加し、コンストラクタの唯一の引数としてドロップアウト確率を渡すだけで済む。学習中、Dropout 層は指定されたドロップアウト確率に従って、前の層の出力（あるいは同値に、次の層への入力）をランダムにドロップする。学習モードでないときは、Dropout 層は単にデータをそのまま通してテストする。

```
class DropoutMLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.Flatten(), nn.LazyLinear(num_hiddens_1), nn.ReLU(),
            nn.Dropout(dropout_1), nn.LazyLinear(num_hiddens_2), nn.ReLU(),
            nn.Dropout(dropout_2), nn.LazyLinear(num_outputs))
```

```
class DropoutMLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr):
```

(continues on next page)

(continued from previous page)

```
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential()
        self.net.add(nn.Dense(num_hiddens_1, activation="relu"),
                     nn.Dropout(dropout_1),
                     nn.Dense(num_hiddens_2, activation="relu"),
                     nn.Dropout(dropout_2),
                     nn.Dense(num_outputs))
        self.net.initialize()
```

```
class DropoutMLP(d2l.Classifier):
    num_hiddens_1: int
    num_hiddens_2: int
    num_outputs: int
    dropout_1: float
    dropout_2: float
    lr: float
    training: bool = True

    @nn.compact
    def __call__(self, X):
        x = nn.relu(nn.Dense(self.num_hiddens_1)(X.reshape((X.shape[0], -1))))
        x = nn.Dropout(self.dropout_1, deterministic=not self.training)(x)
        x = nn.relu(nn.Dense(self.num_hiddens_2)(x))
        x = nn.Dropout(self.dropout_2, deterministic=not self.training)(x)
        return nn.Dense(self.num_outputs)(x)
```

```
class DropoutMLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = tf.keras.models.Sequential([
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(num_hiddens_1, activation=tf.nn.relu),
            tf.keras.layers.Dropout(dropout_1),
            tf.keras.layers.Dense(num_hiddens_2, activation=tf.nn.relu),
            tf.keras.layers.Dropout(dropout_2),
            tf.keras.layers.Dense(num_outputs)])
```

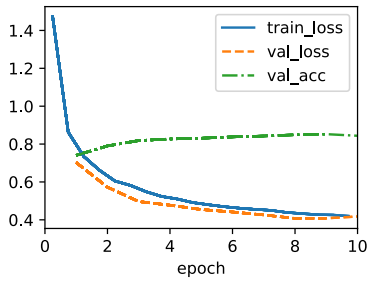
次に、モデルを学習する。

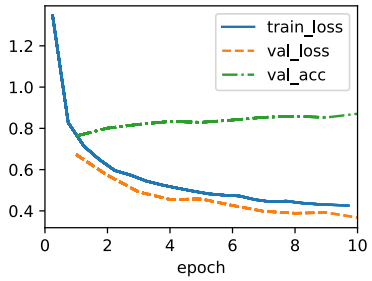
```
model = DropoutMLP(**hparams)
trainer.fit(model, data)
```

```
model = DropoutMLP(**hparams)
trainer.fit(model, data)
```

```
@d2l.add_to_class(d2l.Classifier)  #@save
```

(continues on next page)

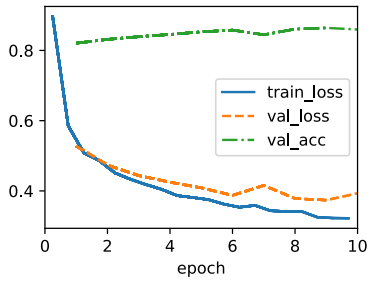




(continued from previous page)

```
@partial(jax.jit, static_argnums=(0, 5))
def loss(self, params, X, Y, state, averaged=True):
    Y_hat = state.apply_fn({'params': params}, *X,
                           mutable=False,  # To be used later (e.g., batch
↪norm)
                           rngs={'dropout': state.dropout_rng})
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    Y = d2l.reshape(Y, (-1,))
    fn = optax.softmax_cross_entropy_with_integer_labels
    # The returned empty dictionary is a placeholder for auxiliary data,
    # which will be used later (e.g., for batch norm)
    return (fn(Y_hat, Y).mean(), {}) if averaged else (fn(Y_hat, Y), {})
```

```
model = DropoutMLP(**hparams)
trainer.fit(model, data)
```



8.6.4 まとめ

次元数や重みベクトルの大きさを制御することに加えて、ドロップアウトは過学習を避けるためのもう 1 つの手段である。多くの場合、これらの手段は組み合わせて使われる。ドロップアウトは学習時にのみ使われることに注意されたい。これは、活性化 h を期待値が h のランダム変数に置き換える。

8.6.5 演習

1. 1 層目と 2 層目のドロップアウト確率を変えるとどうなるか？特に、両方の層の確率を入れ替えるとどうなるか？これらの質問に答える実験を設計し、結果を定量的に述べ、定性的な要点をまとめなさい。
2. エポック数を増やし、ドロップアウトを使った場合と使わない場合の結果を比較しなさい。
3. ドロップアウトを適用した場合としない場合で、各隠れ層の活性化の分散はどれくらいだろうか？この量が時間とともにどのように変化するかを、両方のモデルについてプロットしなさい。
4. なぜドロップアウトは通常テスト時には使われないのだろうか？
5. この節のモデルを例に、ドロップアウトと重み減衰の効果を比較しなさい。ドロップアウトと重み減衰を同時に使うとどうなるか？効果は加算的だろうか？それとも逓減するか（あるいは悪化するか）？互いに打ち消し合いますか？
6. 活性化ではなく、重み行列の個々の重みにドロップアウトを適用するとどうなるか？
7. 標準的なドロップアウト手法とは異なる、各層にランダムノイズを注入する別の手法を考案しなさい。Fashion-MNIST データセットで（固定アーキテクチャに対して）ドロップアウトを上回る方法を開発できるか？

8.7 Kaggle で住宅価格を予測する

ここまでで、深層ネットワークを構築・学習するための基本的な道具を導入し、さらに重み減衰やドロップアウトといった手法でそれらを正則化する方法も学んだ。いよいよ、Kaggle コンペティションに参加して、これまでの知識を実践に移す準備が整った。住宅価格予測コンペティションは、始めるのに最適な題材である。データは比較的一般的で、音声や動画のように特殊なモデルを必要とするような特異な構造は含んでいない。このデータセットは De Cock (2011) によって収集されたもので、2006–2010 年のアイオワ州エイムズの住宅価格を扱っている。これは、Harrison と Rubinfeld (1978) による有名な Boston housing dataset[72] よりもかなり大きく、データ例の数も特徴量の数も多くなって



いる。

この節では、データ前処理、モデル設計、ハイパーパラメータ選択の詳細を順に説明する。実践的なアプローチを通じて、データサイエンティストとしてのキャリアに役立つ直感を身につけてもらえればと思う。

```
%matplotlib inline
from d2l import torch as d2l
import torch
from torch import nn
import pandas as pd
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, autograd, init, np, npx
from mxnet.gluon import nn
import pandas as pd

npx.set_np()
```

```
%matplotlib inline
from d2l import jax as d2l
import jax
from jax import numpy as jnp
import numpy as np
import pandas as pd
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf
import pandas as pd
```

8.7.1 データのダウンロード

本書を通して、さまざまなダウンロード済みデータセットを用いてモデルを学習・評価する。ここでは、zip または tar ファイルをダウンロードして展開するための 2 つのユーティリティ関数を実装する。ここでも、そのようなユーティリティ関数の実装詳細は省略する。

```
def download(url, folder, sha1_hash=None):
    """Download a file to folder and return the local filepath."""

def extract(filename, folder):
    """Extract a zip/tar file into folder."""
```

8.7.2 Kaggle

Kaggle[73] は、機械学習コンペティションを開催する人気のあるプラットフォームである。各コンペティションは 1 つのデータセットを中心に構成されており、多くは賞金を提供する利害関係者によって後援されている。このプラットフォームは、フォーラムや共有コードを通じてユーザー同士の交流を支援し、協力と競争の両方を促進する。ランキング上位を追い求めるあまり暴走し、研究者が本質的な問いを立てることよりも前処理の細部に視野狭窄的に集中してしまうことも少なくないが、一方で、競合手法を直接定量比較できるようにするプラットフォームの客観性には大きな価値がある。さらに、コード共有によって何がうまくいき、何がうまくいかなかったのかを誰もが学べるようになる。Kaggle コンペティションに参加したい場合は、まずアカウント登録が必要である。(Fig. 8.7.1 を参照)。

73

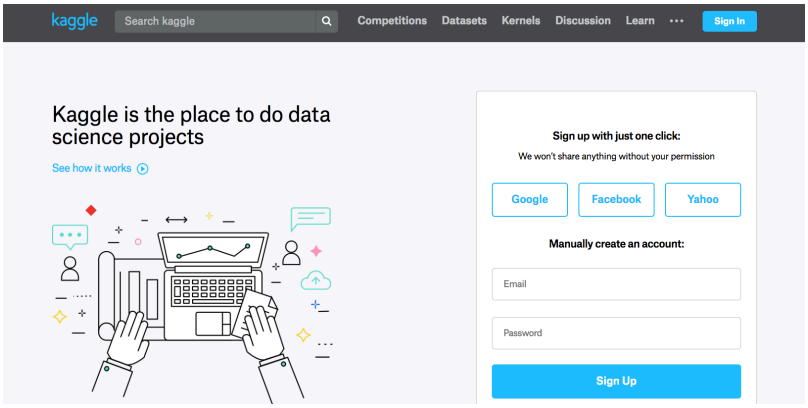


Fig. 8.7.1 The Kaggle website.

Fig. 8.7.2 に示すように、住宅価格予測コンペティションのページでは、データセットを見つけたり（“Data”タブの下）、予測を提出したり、順位を確認したりできる。URL は次のとおりである。

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

8.7.3 データセットへのアクセスと読み込み

コンペティションのデータは訓練セットとテストセットに分かれていることに注意されたい。各レコードには住宅の資産価値と、道路の種類、建築年、屋根の種類、地下室の状態などの属性が含まれる。特徴量にはさまざまなデータ型が含まれている。たとえば、建築年は整数で表され、屋根の種類は離散的なカテゴリ割り当てで表され、その他の特徴量は浮動小数点数で表される。そして、ここで現実の厄介さが出てくる。一部のデータ例ではデータが完全に欠落しており、欠損値は単に“na”として示されている。各住宅の価格は訓練セットにのみ含まれている（コンペティションなのだから当然である）。訓練セットを分割して検証セットを作りたいところであるが、Kaggle に予測をアップロードした後でしか公式テストセット上でモデルを評価できない。Fig. 8.7.2 のコンペ

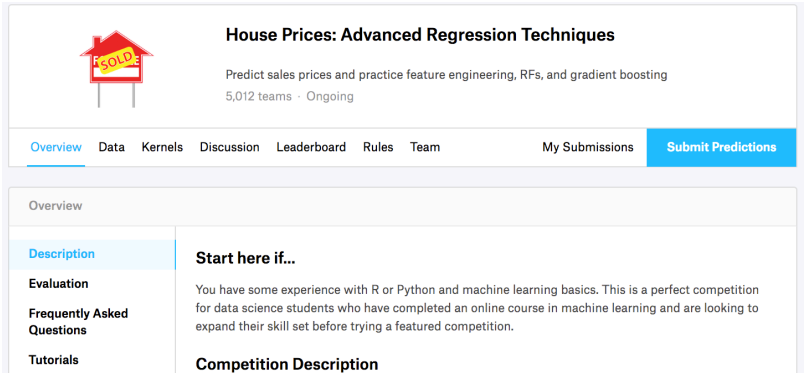


Fig. 8.7.2 The house price prediction competition page.

ティションタブにある“Data”タブには、データをダウンロードするためのリンクがある。

まずは、Section 5.2 で紹介した pandas を使ってデータを読み込み、処理してみよう。便宜上、Kaggle の住宅データセットをダウンロードしてキャッシュできる。このデータセットに対応するファイルがすでにキャッシュディレクトリに存在し、その SHA-1 が sha1_hash と一致する場合、冗長なダウンロードでインターネット回線を無駄にしないよう、キャッシュ済みファイルを使用する。

```
class KaggleHouse(d2l.DataModule):
    def __init__(self, batch_size, train=None, val=None):
        super().__init__()
        self.save_hyperparameters()
        if self.train is None:
            self.raw_train = pd.read_csv(d2l.download(
                d2l.DATA_URL + 'kaggle_house_pred_train.csv', self.root,
                sha1_hash='585e9cc93e70b39160e7921475f9bcd7d31219ce'))
            self.raw_val = pd.read_csv(d2l.download(
                d2l.DATA_URL + 'kaggle_house_pred_test.csv', self.root,
                sha1_hash='fa19780a7b011d9b009e8bff8e99922a8ee2eb90'))
```

訓練データセットには 1460 個のデータ例、80 個の特徴量、1 個のラベルが含まれ、一方で検証データには 1459 個の例と 80 個の特徴量が含まれている。

```
data = KaggleHouse(batch_size=64)
print(data.raw_train.shape)
print(data.raw_val.shape)
```

```
Downloading ../data/kaggle_house_pred_train.csv from http://d2l-data.s3-
↪accelerate.amazonaws.com/kaggle_house_pred_train.csv...
Downloading ../data/kaggle_house_pred_test.csv from http://d2l-data.s3-
↪accelerate.amazonaws.com/kaggle_house_pred_test.csv...
(1460, 81)
(1459, 80)
```

8.7.4 データ前処理

最初の 4 つのデータ例について、最初の 4 つと最後の 2 つの特徴量、およびラベル（SalePrice）を見てみましょう。

```
print(data.raw_train.iloc[:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
0   1          60       RL         65.0       WD        Normal     208500
1   2          20       RL         80.0       WD        Normal     181500
2   3          60       RL         68.0       WD        Normal     223500
3   4          70       RL         60.0       WD       Abnorml     140000
```

各データ例では、最初の特徴量が識別子であることがわかる。これはモデルが各訓練データ例を区別するのに役立ちる。便利ではあるが、予測の目的には何の情報も持たない。したがって、モデルにデータを入力する前に、この特徴量はデータセットから削除する。さらに、さまざまなデータ型が混在しているため、モデリングを始める前にデータ前処理が必要になる。

まず数値特徴量から始めよう。最初にヒューリスティックとして、すべての欠損値を対応する特徴量の平均値で置き換える。その後、すべての特徴量を共通の尺度にそろえるために、データを標準化し、特徴量を平均 0・分散 1 に再スケーリングする:

$$x \leftarrow \frac{x - \mu}{\sigma}, \tag{8.7.1}$$

ここで μ と σ はそれぞれ平均と標準偏差を表す。これにより特徴量（変数）が本当に平均 0・分散 1 になることを確認するには、$E[\frac{x-\mu}{\sigma}] = \frac{\mu-\mu}{\sigma} = 0$ であり、また $E[(x-\mu)^2] = (\sigma^2 + \mu^2) - 2\mu^2 + \mu^2 = \sigma^2$ であることに注意されたい。直感的には、データを標準化する理由は 2 つある。第 1 に、最適化に都合がよいことである。第 2 に、どの特徴量が関連するかを *a priori* に知ることはできないので、ある特徴量に割り当てられた係数を他よりも強く罰したくないからである。

次に離散値を扱う。これには“MSZoning”のような特徴量が含まれる。これらは one-hot エンコーディングで置き換えるこれは、先に多クラスラベルをベクトルに変換したときと同じ方法である (Section 7.1.1 を参照)。たとえば、“MSZoning”は“RL”と“RM”の値を取る。“MSZoning”特徴量を削除すると、値が 0 か 1 の 2 つの新しい指示特徴量“MSZoning_RL”と“MSZoning_RM”が作成される。one-hot エンコーディングに従えば、元の“MSZoning”の値が“RL”なら、“MSZoning_RL”は 1 で“MSZoning_RM”は 0 である。pandas パッケージはこれを自動的に行ってくれる。

```
@d2l.add_to_class(KaggleHouse)
def preprocess(self):
    # Remove the ID and label columns
    label = 'SalePrice'
```

(continues on next page)

(continued from previous page)

```
    features = pd.concat(
        (self.raw_train.drop(columns=['Id', label]),
         self.raw_val.drop(columns=['Id'])))
    # Standardize numerical columns
    numeric_features = features.dtypes[features.dtypes!='object'].index
    features[numeric_features] = features[numeric_features].apply(
        lambda x: (x - x.mean()) / (x.std()))
    # Replace NAN numerical features by 0
    features[numeric_features] = features[numeric_features].fillna(0)
    # Replace discrete features by one-hot encoding
    features = pd.get_dummies(features, dummy_na=True)
    # Save preprocessed features
    self.train = features[:self.raw_train.shape[0]].copy()
    self.train[label] = self.raw_train[label]
    self.val = features[self.raw_train.shape[0]:].copy()
```

この変換によって、特徴量の数が 79 から 331 に増えることがわかる（ID 列とラベル列を除く）。

```
data.preprocess()
data.train.shape
```

```
(1460, 331)
```

8.7.5 誤差尺度

まずは二乗損失を用いた線形モデルを学習してみよう。当然ながら、この線形モデルがコンペティションで勝てる提出につながることはないが、データの中に意味のある情報があるかどうかを確認するための健全性チェックにはなる。ここでランダム予測よりも良い結果が出せないなら、データ処理にバグがある可能性が高いだろう。そして、うまくいくなら、線形モデルはベースラインとして機能し、単純なモデルが最良報告モデルにどれくらい近づけるのかについての直感を与え、より洗練されたモデルからどれほどの改善が期待できるかの目安になる。

住宅価格では、株価と同様に、絶対量よりも相対量のほうが重要である。したがって、絶対誤差 $y - \hat{y}$ よりも相対誤差 $\frac{y-\hat{y}}{y}$ のほうを重視する傾向がある。たとえば、オハイオ州の田舎で住宅価格を推定していて、典型的な住宅の価値が 125,000 ドルであるときに、予測が 100,000 ドル外れたとしたら、おそらくひどい出来である。一方、カリフォルニア州ロスアルトスヒルズで同じだけ外れたとしても、それは驚くほど正確な予測かもしれない（そこでは住宅価格の中央値が 400 万ドルを超える）。

この問題に対処する 1 つの方法は、価格推定の対数における差を測ることである。実際、これはコンペティションが提出物の品質を評価するために用いている公式の誤差尺度でもありる。結局のところ、$|\log y - \log \hat{y}| \leq \delta$ のような小さな値 δ は、$e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$ に対応しる。これにより、予測価格の対数とラベル価格

の対数の間の次の二乗平均平方根誤差が得られる。

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log y_i - \log \hat{y}_i)^2}. \tag{8.7.2}$$

```
@d2l.add_to_class(KaggleHouse)
def get_dataloader(self, train):
    label = 'SalePrice'
    data = self.train if train else self.val
    if label not in data: return
    get_tensor = lambda x: d2l.tensor(x.values.astype(float),
                                      dtype=d2l.float32)
    # Logarithm of prices
    tensors = (get_tensor(data.drop(columns=[label])),  # X
               d2l.reshape(d2l.log(get_tensor(data[label])), (-1, 1)))  # Y
    return self.get_tensorloader(tensors, train)
```

8.7.6 K-分割交差検証

モデル選択 (page 198) で交差検証を導入し、モデル選択の扱いについて議論したことを思い出してほしい。ここではそれを活用して、モデル設計の選択やハイパーパラメータの調整を行う。まず、K-分割交差検証手順におけるデータの i^{th} 分割を返す関数が必要である。これは、i^{th} 区間を検証データとして切り出し、残りを訓練データとして返す。これはデータの扱いとして最も効率的な方法ではないことに注意されたい。データセットがかなり大きい場合には、もっと賢い方法を確実に採るだろう。しかし、この問題は単純なので、そのような追加の複雑さはコードを不必要にわかりにくくしてしまうかもしれない。したがって、ここでは省略しても問題ない。

```
def k_fold_data(data, k):
    rets = []
    fold_size = data.train.shape[0] // k
    for j in range(k):
        idx = range(j * fold_size, (j+1) * fold_size)
        rets.append(KaggleHouse(data.batch_size, data.train.drop(index=idx),
                                data.train.loc[idx]))
    return rets
```

K 回学習したときの平均検証誤差が返される。K-分割交差検証では、これを用いる。

```
def k_fold(trainer, data, k, lr):
    val_loss, models = [], []
    for i, data_fold in enumerate(k_fold_data(data, k)):
        model = d2l.LinearRegression(lr)
        model.board.yscale='log'
        if i != 0: model.board.display = False
        trainer.fit(model, data_fold)
        val_loss.append(float(model.board.data['val_loss'][-1].y))
```

(continues on next page)

(continued from previous page)

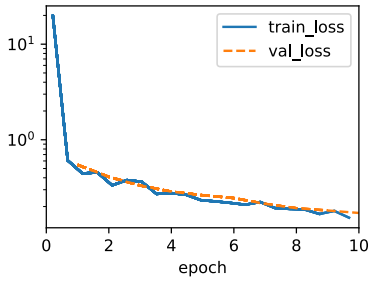
```
        models.append(model)
    print(f'average validation log mse = {sum(val_loss)/len(val_loss)}')
    return models
```

8.7.7 モデル選択

この例では、調整していないハイパーパラメータの組を選び、モデルの改善は読者に委ねる。適切な選択を見つけるには、最適化する変数の数に応じて時間がかかることがある。十分に大きなデータセットと通常の種類のハイパーパラメータであれば、K-分割交差検証は複数回の試行に対してかなり頑健である傾向がある。しかし、非現実的に多くの選択肢を試すと、検証性能が真の誤差をもはや代表しなくなるかもしれない。

```
trainer = d2l.Trainer(max_epochs=10)
models = k_fold(trainer, data, k=5, lr=0.01)
```

```
average validation log mse = 0.17631081134080887
```



ときには、あるハイパーパラメータ集合に対する訓練誤差の数が非常に少ないのに、K-分割交差検証での誤差数はかなり大きくなることがある。これは過学習していることを示している。学習中は、この 2 つの数値の両方を監視したいところである。過学習が小さいということは、データがより強力なモデルを支えられる可能性を示しているかもしれない。大きな過学習は、正則化手法を取り入れることで改善できることを示唆しているかもしれない。

8.7.8 Kaggle への予測提出

良いハイパーパラメータの選び方がわかったので、K 個のモデルすべてによるテストセット上の予測の平均を計算してみよう。予測を csv ファイルに保存しておくと、Kaggle への結果アップロードが簡単になる。次のコードは `submission.csv` というファイルを生成する。

```
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    preds = [model(d2l.tensor(data.val.values.astype(float), dtype=d2l.
↪float32))
             for model in models]
if tab.selected('jax'):
    preds = [model.apply({'params': trainer.state.params},
             d2l.tensor(data.val.values.astype(float), dtype=d2l.float32))
             for model in models]
# Taking exponentiation of predictions in the logarithm scale
ensemble_preds = d2l.reduce_mean(d2l.exp(d2l.concat(preds, 1)), 1)
submission = pd.DataFrame({'Id':data.raw_val.Id,
                           'SalePrice':d2l.numpy(ensemble_preds)})
submission.to_csv('submission.csv', index=False)
```

次に、Fig. 8.7.3 に示すように、Kaggle に予測を提出し、テストセット上の実際の住宅価格（ラベル）とどの程度一致しているかを確認できる。手順はとても簡単である。

- Kaggle のウェブサイトにログインし、住宅価格予測コンペティションのページを開く。
- “Submit Predictions”または“Late Submission”ボタンをクリックする。
- ページ下部の点線枠内にある“Upload Submission File”ボタンをクリックし、アップロードしたい予測ファイルを選択する。
- ページ下部の“Make Submission”ボタンをクリックして結果を表示する。

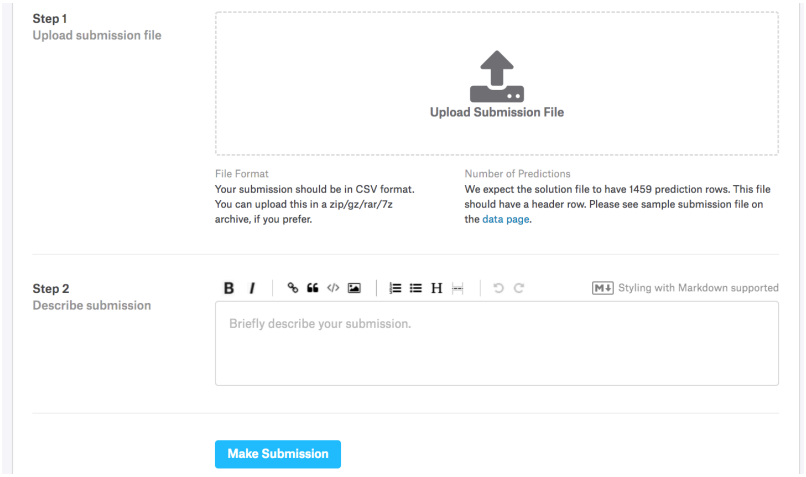


Fig. 8.7.3 Submitting data to Kaggle.

8.7.9 まとめと考察

実データにはしばしばさまざまなデータ型が混在しており、前処理が必要である。実数値データを平均 0・分散 1 に再スケーリングするのは良いデフォルトである。欠損値を平均値で置き換えるのも同様である。さらに、カテゴリ特徴量

を指示特徴量に変換すると、それらを one-hot ベクトルとして扱えるようになりる。絶対誤差よりも相対誤差を重視する傾向がある場合には、予測の対数における差を測ることができる。モデルを選択し、ハイパーパラメータを調整するには、K-分割交差検証を使える。

8.7.10 演習

1. この節の予測を Kaggle に提出してみしよう。どの程度の成績でしたか？
2. 欠損値を平均値で置き換えるのは常に良い考えだろうか？ヒント: 値がランダムに欠損していない状況を構成できるか？
3. K-分割交差検証によってハイパーパラメータを調整し、スコアを改善ししよう。
4. モデルを改善することでスコアを改善しましょう（たとえば、層、重み減衰、ドロップアウト）。
5. この節で行ったように連続数値特徴量を標準化しないとどうなるか？

9 ビルダーのためのガイド

巨大なデータセットや強力なハードウェアと並んで、優れたソフトウェアツールは深層学習の急速な進歩において不可欠な役割を果たしてきた。2007 年に画期的な Theano ライブラリが公開されて以来、柔軟なオープンソースツールは研究者がモデルを素早く試作することを可能にし、標準的なコンポーネントを再利用する際の反復作業を避けつつ、低レベルの修正を行う能力は維持してきた。時が経つにつれて、深層学習のライブラリはますます粗い抽象化を提供する方向へ進化してきた。半導体設計者がトランジスタの指定から論理回路へ、そしてコードを書くことへと移っていったように、ニューラルネットワーク研究者も個々の人工ニューロンの振る舞いを考える段階から、ネットワークを層全体として捉える段階へと移り、現在ではさらにずっと粗い ブロックを念頭に置いてアーキテクチャを設計することが多くなっている。

これまでに、いくつかの基本的な機械学習の概念を紹介し、完全に機能する深層学習モデルへと段階的に進んできた。前章では、MLP の各構成要素をゼロから実装し、さらに高レベル API を活用して同じモデルを手軽に展開する方法まで示した。そこまでをあれほど速く進めるために、私たちはライブラリを 利用した一方で、それらが どのように動作するかという、より高度な詳細には触れなかった。この章では、その幕を引きはがし、深層学習計算の主要な構成要素、すなわちモデル構築、パラメータへのアクセスと初期化、カスタム層とブロックの設計、モデルのディスクへの読み書き、そして GPU を活用して劇的な高速化を実現する方法について、さらに深く掘り下げる。これらの知識によって、あなたは エンドユーザーから パワーユーザーへと進み、成熟した深層学習ライブラリの利点を享受しながら、自分で考案したものを含む、より複雑なモデルを実装する柔軟性も保つために必要な道具を手に入れることができる！この章では新しいモデルやデータセットは導入しないが、続く高度なモデリングの章では、これらの技術に大きく依存する。

9.1 層とモジュール

ニューラルネットワークを最初に導入したとき、私たちは単一の出力をもつ線形モデルに焦点を当てた。ここでは、モデル全体がたった 1 つのニューロンだけで構成されている。単一のニューロンは、(i) ある入力集合を受け取り、(ii) それに対応するスカラー出力を生成し、(iii) 関心のある目的関数を最適化するために更新可能な関連パラメータの集合をもつ、ことに注意してほしい。その後、

複数の出力をもつネットワークについて考え始めると、ベクトル化された算術を活用して、ニューロンの層全体を表現できるようになった。個々のニューロンと同様に、層も (i) 入力の集合を受け取り、(ii) 対応する出力を生成し、(iii) 調整可能なパラメータの集合によって記述される。softmax 回帰を扱ったときには、1 つの層そのものがモデルであった。しかし、その後 MLP を導入した後でも、モデルは同じ基本構造を保っていると考えることができた。

興味深いことに、MLP では、モデル全体とその構成要素である層の両方がこの構造を共有している。モデル全体は生の入力（特徴量）を受け取り、出力（予測）を生成し、パラメータ（すべての構成層のパラメータを合わせたもの）を持つ。同様に、各個別の層は入力（前の層から供給される）を受け取り、出力（次の層への入力）を生成し、さらに、次の層から逆向きに流れてくる信号に応じて更新される調整可能なパラメータの集合を持つ。

ニューロン、層、モデルで十分な抽象化が得られているように思えるかもしれないが、実際には、個々の層より大きく、モデル全体よりは小さい構成要素について述べると便利なことがよくある。たとえば、コンピュータビジョンで非常に人気のある ResNet-152 アーキテクチャは、数百もの層を持っている。これらの層は、繰り返し現れる層のグループのパターンから構成されている。こうしたネットワークを 1 層ずつ実装するのは面倒になりがちである。この懸念は単なる仮説ではない。こうした設計パターンは実際によく見られる。上で述べた ResNet アーキテクチャは、認識と検出の両方で 2015 年の ImageNet および COCO のコンピュータビジョン競技会を制し (He *et al.*, 2016) 、今なお多くの視覚タスクで定番のアーキテクチャである。層がさまざまな繰り返しパターンで配置された同様のアーキテクチャは、自然言語処理や音声を含む他の分野でも今や至るところに見られる。

こうした複雑なネットワークを実装するために、ニューラルネットワークのモジュールという概念を導入する。モジュールは単一の層、複数の層からなる構成要素、あるいはモデル全体そのものを表すことができる！モジュール抽象化を用いる利点の 1 つは、それらをより大きな成果物へと組み合わせられることである。しかも、その組み合わせはしばしば再帰的に行える。これは Fig. 9.1.1 に示されている。任意の複雑さをもつモジュールを必要に応じて生成するコードを定義することで、驚くほど簡潔なコードで複雑なニューラルネットワークを実装できる。

プログラミングの観点では、モジュールはクラスとして表現される。そのサブクラスはすべて、入力を出力へ変換する順伝播メソッドを定義しなければならず、必要なパラメータを保存しなければならない。なお、モジュールの中にはパラメータをまったく必要としないものもある。最後に、モジュールは勾配を計算するための逆伝播メソッドを備えていなければならない。幸いなことに、自動微分 (Section 5.5 で導入) によって提供される裏方の魔法のおかげで、自分自身のモジュールを定義するときに私たちが気にする必要があるのはパラメータと順伝播メソッドだけである。

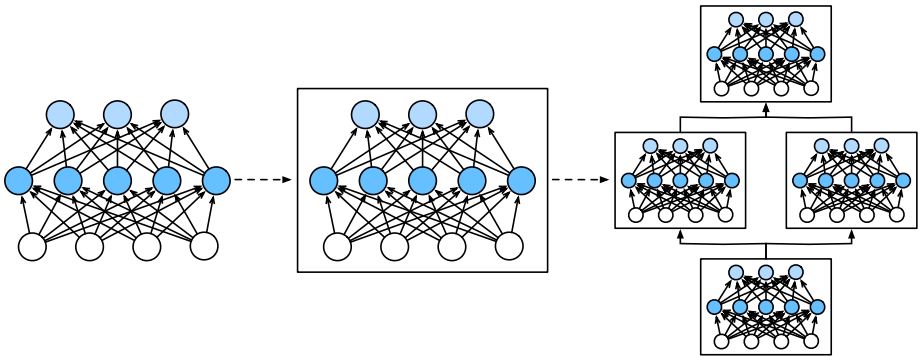


Fig. 9.1.1 複数の層がモジュールにまとめられ、より大きなモデルの繰り返しパターンを形成する。

```
import torch
from torch import nn
from torch.nn import functional as F
```

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from typing import List
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
import tensorflow as tf
```

まず、MLP を実装するために使ったコードを再確認しよう (Section 8.1)。次のコードは、256 ユニットと ReLU 活性化をもつ全結合隠れ層 1 つと、10 ユニットをもつ全結合出力層 1 つ（活性化関数なし）からなるネットワークを生成する。

```
net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))

X = torch.rand(2, 20)
net(X).shape
```

```
torch.Size([2, 10])
```

```
net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()

X = np.random.uniform(size=(2, 20))
net(X).shape
```

```
[07:07:55] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(2, 10)
```

```
net = nn.Sequential([nn.Dense(256), nn.relu, nn.Dense(10)])

# get_key is a d2l saved function returning jax.random.PRNGKey(random_seed)
X = jax.random.uniform(d2l.get_key(), (2, 20))
params = net.init(d2l.get_key(), X)
net.apply(params, X).shape
```

```
(2, 10)
```

```
net = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    tf.keras.layers.Dense(10),
])

X = tf.random.uniform((2, 20))
net(X).shape
```

```
TensorShape([2, 10])
```

この例では、nn.Sequential をインスタンス化し、実行される順序で層を引数として渡すことでモデルを構築した。要するに、nn.Sequential は特別な種類の Module を定義するものであり、PyTorch におけるモジュールを表すクラスである。これは構成要素である Module の順序付きリストを保持する。2 つの全結合層はいずれも Linear クラスのインスタンスであり、それ自体が Module のサブクラスであることに注意してほしい。順伝播（forward）メソッドも非常に単純である。リスト内の各モジュールを連結し、それぞれの出力を次の入力として渡していく。なお、これまで私たちはモデルの出力を得るために net(X) という構文でモデルを呼び出してきた。これは実際には net.__call__(X) の省略形にすぎない。

9.1.1 カスタムモジュール

モジュールの仕組みを理解する最も簡単な方法は、自分で 1 つ実装してみることかもしれない。その前に、各モジュールが提供しなければならない基本機能を簡単にまとめておこう。

1. 順伝播メソッドの引数として入力データを受け取る。
2. 順伝播メソッドが値を返すことで出力を生成する。出力の形状は入力と異なっていてもよい。たとえば、上のモデルの最初の全結合層は任意次元の入力を受け取るが、256 次元の出力を返す。
3. 出力の入力に関する勾配を計算する。これは逆伝播メソッドを通じてアクセスできる。通常、これは自動的に行われる。
4. 順伝播計算を実行するために必要なパラメータを保存し、それらへアクセスできるようにする。
5. 必要に応じてモデルパラメータを初期化する。

次のスニペットでは、隠れユニット 256 個の隠れ層 1 つと 10 次元の出力層 1 つをもつ MLP に対応するモジュールをゼロから記述する。以下の `MLP` クラスは、モジュールを表すクラスを継承していることに注意してほしい。親クラスのメソッドに大きく依存し、独自に実装するのはコンストラクタ（Python では `__init__` メソッド）と順伝播メソッドだけである。

```
class MLP(nn.Module):
    def __init__(self):
        # Call the constructor of the parent class nn.Module to perform
        # the necessary initialization
        super().__init__()
        self.hidden = nn.LazyLinear(256)
        self.out = nn.LazyLinear(10)

    # Define the forward propagation of the model, that is, how to return the
    # required model output based on the input X
    def forward(self, X):
        return self.out(F.relu(self.hidden(X)))
```

```
class MLP(nn.Block):
    def __init__(self):
        # Call the constructor of the MLP parent class nn.Block to perform
        # the necessary initialization
        super().__init__()
        self.hidden = nn.Dense(256, activation='relu')
        self.out = nn.Dense(10)

    # Define the forward propagation of the model, that is, how to return the
    # required model output based on the input X
    def forward(self, X):
        return self.out(self.hidden(X))
```

```
class MLP(nn.Module):
    def setup(self):
        # Define the layers
        self.hidden = nn.Dense(256)
        self.out = nn.Dense(10)

    # Define the forward propagation of the model, that is, how to return the
    # required model output based on the input X
    def __call__(self, X):
        return self.out(nn.relu(self.hidden(X)))
```

```
class MLP(tf.keras.Model):
    def __init__(self):
        # Call the constructor of the parent class tf.keras.Model to perform
        # the necessary initialization
        super().__init__()
        self.hidden = tf.keras.layers.Dense(units=256, activation=tf.nn.relu)
        self.out = tf.keras.layers.Dense(units=10)

    # Define the forward propagation of the model, that is, how to return the
    # required model output based on the input X
    def call(self, X):
        return self.out(self.hidden((X)))
```

まず順伝播メソッドに注目しよう。これは X を入力として受け取り、活性化関数を適用した隠れ表現を計算し、ロジットを出力することがわかる。この MLP 実装では、両方の層がインスタンス変数である。これが妥当である理由を理解するために、net1 と net2 という 2 つの MLP をインスタンス化し、それぞれを異なるデータで学習させる場面を想像してほしい。当然、それらは 2 つの異なる学習済みモデルを表すはずである。

私たちはコンストラクタの中で MLP の層をインスタンス化しその後、順伝播メソッドが呼ばれるたびにこれらの層を呼び出す。いくつか重要な点に注意してほしい。まず、カスタマイズした __init__ メソッドは super().__init__() を通じて親クラスの __init__ メソッドを呼び出しており、これにより、多くのモジュールに共通する定型コードを何度も書き直す手間を省いている。次に、2 つの全結合層をインスタンス化し、それらを self.hidden と self.out に代入している。新しい層を実装しない限り、逆伝播メソッドやパラメータ初期化について心配する必要はない。これらのメソッドはシステムが自動的に生成する。試してみよう。

```
net = MLP()
if tab.selected('mxnet'):
    net.initialize()
net(X).shape
```

```
torch.Size([2, 10])
```

```
net = MLP()
if tab.selected('mxnet'):
    net.initialize()
net(X).shape
```

```
(2, 10)
```

```
net = MLP()
params = net.init(d2l.get_key(), X)
net.apply(params, X).shape
```

```
(2, 10)
```

```
net = MLP()
if tab.selected('mxnet'):
    net.initialize()
net(X).shape
```

```
TensorShape([2, 10])
```

モジュール抽象化の大きな利点は、その柔軟性である。モジュールをサブクラス化して、層（たとえば全結合層クラス）、モデル全体（上の MLP クラスのようなもの）、あるいは中間的な複雑さをもつさまざまな構成要素を作れる。この柔軟性は、今後の章を通じて活用していく。たとえば、畳み込みニューラルネットワークを扱うときなどである。

9.1.2 Sequential モジュール

ここで Sequential クラスの仕組みをもう少し詳しく見てみよう。Sequential は他のモジュールを数珠つなぎにするために設計されていたことを思い出してほしい。独自の簡略版 MySequential を作るには、次の 2 つの重要なメソッドを定義すれば十分である。

1. モジュールを 1 つずつリストに追加するメソッド。
2. 追加された順序と同じ順でモジュールの連鎖を入力に通す順伝播メソッド。

次の MySequential クラスは、デフォルトの Sequential クラスと同じ機能を提供する。

```
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
            self.add_module(str(idx), module)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, X):
        for module in self.children():
            X = module(X)
        return X
```

```
class MySequential(nn.Block):
    def add(self, block):
        # Here, block is an instance of a Block subclass, and we assume that
        # it has a unique name. We save it in the member variable _children of
        # the Block class, and its type is OrderedDict. When the MySequential
        # instance calls the initialize method, the system automatically
        # initializes all members of _children
        self._children[block.name] = block

    def forward(self, X):
        # OrderedDict guarantees that members will be traversed in the order
        # they were added
        for block in self._children.values():
            X = block(X)
        return X
```

```
class MySequential(nn.Module):
    modules: List

    def __call__(self, X):
        for module in self.modules:
            X = module(X)
        return X
```

```
class MySequential(tf.keras.Model):
    def __init__(self, *args):
        super().__init__()
        self.modules = args

    def call(self, X):
        for module in self.modules:
            X = module(X)
        return X
```

`__init__` メソッドでは、`add_modules` メソッドを呼び出してすべてのモジュールを追加する。これらのモジュールは後で `children` メソッドからアクセスできる。このようにしてシステムは追加されたモジュールを把握し、各モジュールのパラメータを適切に初期化する。

`MySequential` の順伝播メソッドが呼び出されると、追加された各モジュールが追加された順序で実行される。これで、`MySequential` クラスを使って MLP を再実装できる。

```
net = MySequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))
net(X).shape
```

```
torch.Size([2, 10])
```

```
net = MySequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(X).shape
```

```
(2, 10)
```

```
net = MySequential([nn.Dense(256), nn.relu, nn.Dense(10)])
params = net.init(d2l.get_key(), X)
net.apply(params, X).shape
```

```
(2, 10)
```

```
net = MySequential(
    tf.keras.layers.Dense(units=256, activation=tf.nn.relu),
    tf.keras.layers.Dense(10))
net(X).shape
```

```
TensorShape([2, 10])
```

MySequential のこの使い方は、以前 Sequential クラスについて書いたコード (Section 8.1 で説明) とまったく同じであることに注意してほしい。

9.1.3 順伝播メソッド内でコードを実行する

Sequential クラスはモデル構築を簡単にし、自分でクラスを定義しなくても新しいアーキテクチャを組み立てられるようにしてくれる。しかし、すべてのアーキテクチャが単純な数珠つなぎとは限らない。より高い柔軟性が必要な場合には、独自のブロックを定義したくなる。たとえば、順伝播メソッドの中で Python の制御フローを実行したいことがあるだろう。さらに、あらかじめ定義されたニューラルネットワーク層に頼るだけでなく、任意の数学演算を行いたいこともある。

これまでのところ、ネットワーク内のすべての演算はネットワークの活性化値とそのパラメータに対して作用してきた。しかし時には、前の層の結果でも更新可能なパラメータでもない項を組み込みたいことがある。これらを定数パラメータと呼ぶ。たとえば、$f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}$ という関数を計算する層が欲しいと

しよう。ここで $\mathbf{x}$ は入力、$\mathbf{w}$ はパラメータ、c は最適化中に更新されない指定の定数である。これを次のように FixedHiddenMLP クラスとして実装する。

```
class FixedHiddenMLP(nn.Module):
    def __init__(self):
        super().__init__()
        # Random weight parameters that will not compute gradients and
        # therefore keep constant during training
        self.rand_weight = torch.rand((20, 20))
        self.linear = nn.LazyLinear(20)

    def forward(self, X):
        X = self.linear(X)
        X = F.relu(X @ self.rand_weight + 1)
        # Reuse the fully connected layer. This is equivalent to sharing
        # parameters with two fully connected layers
        X = self.linear(X)
        # Control flow
        while X.abs().sum() > 1:
            X /= 2
        return X.sum()
```

```
class FixedHiddenMLP(nn.Block):
    def __init__(self):
        super().__init__()
        # Random weight parameters created with the get_constant method
        # are not updated during training (i.e., constant parameters)
        self.rand_weight = self.params.get_constant(
            'rand_weight', np.random.uniform(size=(20, 20)))
        self.dense = nn.Dense(20, activation='relu')

    def forward(self, X):
        X = self.dense(X)
        # Use the created constant parameters, as well as the relu and dot
        # functions
        X = npx.relu(np.dot(X, self.rand_weight.data()) + 1)
        # Reuse the fully connected layer. This is equivalent to sharing
        # parameters with two fully connected layers
        X = self.dense(X)
        # Control flow
        while np.abs(X).sum() > 1:
            X /= 2
        return X.sum()
```

```
class FixedHiddenMLP(nn.Module):
    # Random weight parameters that will not compute gradients and
    # therefore keep constant during training
    rand_weight: jnp.array = jax.random.uniform(d2l.get_key(), (20, 20))

    def setup(self):
        self.dense = nn.Dense(20)

    def __call__(self, X):
        X = self.dense(X)
```

(continues on next page)

(continued from previous page)

```
        X = nn.relu(X @ self.rand_weight + 1)
        # Reuse the fully connected layer. This is equivalent to sharing
        # parameters with two fully connected layers
        X = self.dense(X)
        # Control flow
        while jnp.abs(X).sum() > 1:
            X /= 2
        return X.sum()
```

```
class FixedHiddenMLP(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.flatten = tf.keras.layers.Flatten()
        # Random weight parameters created with tf.constant are not updated
        # during training (i.e., constant parameters)
        self.rand_weight = tf.constant(tf.random.uniform((20, 20)))
        self.dense = tf.keras.layers.Dense(20, activation=tf.nn.relu)

    def call(self, inputs):
        X = self.flatten(inputs)
        # Use the created constant parameters, as well as the relu and
        # matmul functions
        X = tf.nn.relu(tf.matmul(X, self.rand_weight) + 1)
        # Reuse the fully connected layer. This is equivalent to sharing
        # parameters with two fully connected layers
        X = self.dense(X)
        # Control flow
        while tf.reduce_sum(tf.math.abs(X)) > 1:
            X /= 2
        return tf.reduce_sum(X)
```

このモデルでは、重み（self.rand_weight）がインスタンス化時にランダムに初期化され、その後は定数となる隠れ層を実装している。この重みはモデルパラメータではないため、逆伝播によって更新されることはない。その後、ネットワークはこの「固定」層の出力を全結合層へ通す。

出力を返す前に、私たちのモデルは少し変わったことをした。ℓ_1 ノルムが 1 より大きいかどうかを条件にした while ループを回し、条件を満たすまで出力ベクトルを 2 で割り続けた。最後に、X の各要素の和を返した。私たちの知る限り、標準的なニューラルネットワークでこの操作を行うものはない。この特定の操作が実世界のどんなタスクにも役立つとは限らない。ここでの目的は、任意のコードをニューラルネットワーク計算の流れに組み込む方法を示すことだけである。

```
net = FixedHiddenMLP()
if tab.selected('mxnet'):
    net.initialize()
net(X)
```

```
tensor(0.0036, grad_fn=<SumBackward0>)
```

```
net = FixedHiddenMLP()
if tab.selected('mxnet'):
    net.initialize()
net(X)
```

```
array(0.52637565)
```

```
net = FixedHiddenMLP()
params = net.init(d2l.get_key(), X)
net.apply(params, X)
```

```
Array(-0.13264906, dtype=float32)
```

```
net = FixedHiddenMLP()
if tab.selected('mxnet'):
    net.initialize()
net(X)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=0.8326873>
```

さまざまなモジュールの組み立て方を組み合わせて使うこともできる。次の例では、モジュールをいくつか創造的な方法で入れ子にしている。

```
class NestMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.LazyLinear(64), nn.ReLU(),
                                 nn.LazyLinear(32), nn.ReLU())
        self.linear = nn.LazyLinear(16)

    def forward(self, X):
        return self.linear(self.net(X))

chimera = nn.Sequential(NestMLP(), nn.LazyLinear(20), FixedHiddenMLP())
chimera(X)
```

```
tensor(-0.0192, grad_fn=<SumBackward0>)
```

```
class NestMLP(nn.Block):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.net = nn.Sequential()
```

(continues on next page)

(continued from previous page)

```
        self.net.add(nn.Dense(64, activation='relu'),
                     nn.Dense(32, activation='relu'))
        self.dense = nn.Dense(16, activation='relu')

    def forward(self, X):
        return self.dense(self.net(X))

chimera = nn.Sequential()
chimera.add(NestMLP(), nn.Dense(20), FixedHiddenMLP())
chimera.initialize()
chimera(X)
```

```
array(0.97720534)
```

```
class NestMLP(nn.Module):
    def setup(self):
        self.net = nn.Sequential([nn.Dense(64), nn.relu,
                                  nn.Dense(32), nn.relu])
        self.dense = nn.Dense(16)

    def __call__(self, X):
        return self.dense(self.net(X))


chimera = nn.Sequential([NestMLP(), nn.Dense(20), FixedHiddenMLP()])
params = chimera.init(d2l.get_key(), X)
chimera.apply(params, X)
```

```
Array(-0.05416024, dtype=float32)
```

```
class NestMLP(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.net = tf.keras.Sequential()
        self.net.add(tf.keras.layers.Dense(64, activation=tf.nn.relu))
        self.net.add(tf.keras.layers.Dense(32, activation=tf.nn.relu))
        self.dense = tf.keras.layers.Dense(16, activation=tf.nn.relu)

    def call(self, inputs):
        return self.dense(self.net(inputs))

chimera = tf.keras.Sequential()
chimera.add(NestMLP())
chimera.add(tf.keras.layers.Dense(20))
chimera.add(FixedHiddenMLP())
chimera(X)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=0.551186>
```

9.1.4 要約

個々の層はモジュールになり得る。多くの層が 1 つのモジュールを構成できる。多くのモジュールが 1 つのモジュールを構成できる。

モジュールはコードを含むことができる。モジュールは、パラメータ初期化や逆伝播を含む多くの雑務を引き受けてくれる。層やモジュールの順次連結は `Sequential` モジュールによって処理される。

9.1.5 演習

1. `MySequential` を Python のリストにモジュールを保存するよう変更すると、どのような問題が起こるか？
2. 2 つのモジュール、たとえば `net1` と `net2` を引数に取り、順伝播で両方のネットワークの連結出力を返すモジュールを実装せよ。これは並列モジュールとも呼ばれる。
3. 同じネットワークの複数インスタンスを連結したいとする。同じモジュールの複数インスタンスを生成するファクトリ関数を実装し、それを使ってより大きなネットワークを構築せよ。

9.2 パラメータ管理

アーキテクチャを選び、ハイパーパラメータを設定したら、次は学習ループに進む。ここでの目標は、損失関数を最小化するパラメータ値を見つけることである。学習後には、将来の予測を行うためにこれらのパラメータが必要になる。さらに、場合によってはパラメータを取り出して、別の文脈で再利用したり、モデルをディスクに保存して他のソフトウェアで実行できるようにしたり、あるいは科学的理解を得ることを期待して調べたりしたいこともある。

ほとんどの場合、パラメータがどのように宣言され、操作されるかという細かな詳細は気にせず、深層学習フレームワークに重い処理を任せることができる。しかし、標準的な層を積み重ねたアーキテクチャから離れると、パラメータの宣言や操作の細部に踏み込む必要が出てくることがある。この節では、次の内容を扱う。

- デバッグ、診断、可視化のためのパラメータへのアクセス。
- 異なるモデル構成要素間でのパラメータ共有。

```
import torch
from torch import nn
```

```
from mxnet import init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
import tensorflow as tf
```

まずは隠れ層が 1 つの MLP に注目する。

```
net = nn.Sequential(nn.LazyLinear(8),
                    nn.ReLU(),
                    nn.LazyLinear(1))

X = torch.rand(size=(2, 4))
net(X).shape
```

```
torch.Size([2, 1])
```

```
net = nn.Sequential()
net.add(nn.Dense(8, activation='relu'))
net.add(nn.Dense(1))
net.initialize()  # Use the default initialization method

X = np.random.uniform(size=(2, 4))
net(X).shape
```

```
[07:06:02] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(2, 1)
```

```
net = nn.Sequential([nn.Dense(8), nn.relu, nn.Dense(1)])

X = jax.random.uniform(d2l.get_key(), (2, 4))
params = net.init(d2l.get_key(), X)
net.apply(params, X).shape
```

```
(2, 1)
```

```
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(4, activation=tf.nn.relu),
    tf.keras.layers.Dense(1),
])

X = tf.random.uniform((2, 4))
net(X).shape
```

```
TensorShape([2, 1])
```

9.2.1 パラメータへのアクセス

まずは、すでに知っているモデルからパラメータにアクセスする方法を見ていこう。

モデルが Sequential クラスを使って定義されている場合、まずモデルをリストのようにインデックス指定して任意の層にアクセスできる。各層のパラメータは、その属性に簡単に格納されている。

次のようにして、2 番目の全結合層のパラメータを調べられる。

```
net[2].state_dict()
```

```
OrderedDict([('weight',
              tensor([[-0.0915,  0.2323,  0.0672,  0.0951,  0.0561,  0.2846,␣
↪0.3074, -0.2919]])),
             ('bias', tensor([0.2610]))])
```

```
net[1].params
```

```
dense1_ (
  Parameter dense1_weight (shape=(1, 8), dtype=float32)
  Parameter dense1_bias (shape=(1,), dtype=float32)
)
```

```
params['params']['layers_2']
```

```
FrozenDict({
    kernel: Array([[-0.5666766 ],
           [-0.2169885 ],
           [ 0.37719882],
```

(continues on next page)

(continued from previous page)

```
           [-0.2976346 ],
           [ 0.15896064],
           [ 0.72404635],
           [-0.44518152],
           [-0.08496901]], dtype=float32),
    bias: Array([0.], dtype=float32),
})
```

```
net.layers[2].weights
```

```
[<tf.Variable 'dense_1/kernel:0' shape=(4, 1) dtype=float32, numpy=
 array([[ 0.2972461],
        [ 0.2490077],
        [-0.9411764],
        [ 1.0483162]], dtype=float32)>,
 <tf.Variable 'dense_1/bias:0' shape=(1,) dtype=float32, numpy=array([0.],␣
↪dtype=float32)>]
```

この全結合層には 2 つのパラメータが含まれており、それぞれその層の重みとバイアスに対応している。

対象を絞ったパラメータ

各パラメータはパラメータクラスのインスタンスとして表現されることに注意してほしい。パラメータを使って何か有用なことをするには、まずその内部の数値を取り出す必要がある。その方法はいくつかある。より簡単なものもあれば、より一般的なものもある。次のコードは、2 番目のニューラルネットワーク層からバイアスを取り出す。これはパラメータクラスのインスタンスを返し、さらにそのパラメータの値にアクセスする。

```
type(net[2].bias), net[2].bias.data
```

```
(torch.nn.parameter.Parameter, tensor([0.2610]))
```

```
type(net[1].bias), net[1].bias.data()
```

```
(mxnet.gluon.parameter.Parameter, array([0.]))
```

```
bias = params['params']['layers_2']['bias']
type(bias), bias
```

```
(jaxlib.xla_extension.ArrayImpl, Array([0.], dtype=float32))
```

```
type(net.layers[2].weights[1]), tf.convert_to_tensor(net.layers[2].weights[1])
```

```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 <tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.], dtype=float32)>)
```

パラメータは、値、勾配、その他の情報を含む複雑なオブジェクトである。そのため、値を明示的に要求する必要がある。

値に加えて、各パラメータからは勾配にもアクセスできる。このネットワークではまだ逆伝播を呼び出していないため、初期状態のままである。

```
net[2].weight.grad == None
```

```
True
```

```
net[1].weight.grad()
```

```
array([[0., 0., 0., 0., 0., 0., 0., 0.]])
```

```
jax.tree_util.tree_map(lambda x: x.shape, params)
```

```
FrozenDict({
    params: {
        layers_0: {
            bias: (8,),
            kernel: (4, 8),
        },
        layers_2: {
            bias: (1,),
            kernel: (8, 1),
        },
    },
})
```

```
net.get_weights()
```

```
[array([[ 0.8407554 , -0.70623875, -0.5730775 ,  0.19883841],
        [ 0.22767669, -0.00824153, -0.27324384, -0.05706435],
        [ 0.66750854,  0.72794133, -0.57247126, -0.4776808 ],
        [ 0.35080236,  0.47631615, -0.53025633, -0.19356781]],
       dtype=float32),
 array([0., 0., 0., 0.], dtype=float32),
 array([[ 0.2972461],
        [ 0.2490077],
```

(continues on next page)

(continued from previous page)

```
        [-0.9411764],
        [ 1.0483162]], dtype=float32),
 array([0.], dtype=float32)]
```

すべてのパラメータを一度に

すべてのパラメータに対して操作を行う必要があるとき、1 つずつアクセスするのは面倒になりがちである。特に、より複雑な、たとえば入れ子になったモジュールを扱う場合には、各サブモジュールのパラメータを取り出すためにツリー全体を再帰的にたどる必要があるため、状況はさらに扱いにくくなる。以下では、すべての層のパラメータにアクセスする方法を示す。

```
[(name, param.shape) for name, param in net.named_parameters()]
```

```
[('0.weight', torch.Size([8, 4])),
 ('0.bias', torch.Size([8])),
 ('2.weight', torch.Size([1, 8])),
 ('2.bias', torch.Size([1]))]
```

```
net.collect_params()
```

```
sequential0_ (
  Parameter dense0_weight (shape=(8, 4), dtype=float32)
  Parameter dense0_bias (shape=(8,), dtype=float32)
  Parameter dense1_weight (shape=(1, 8), dtype=float32)
  Parameter dense1_bias (shape=(1,), dtype=float32)
)
```

```
# We need to give the shared layer a name so that we can refer to its
# parameters
shared = nn.Dense(8)
net = nn.Sequential([nn.Dense(8), nn.relu,
                     shared, nn.relu,
                     shared, nn.relu,
                     nn.Dense(1)])

params = net.init(jax.random.PRNGKey(d2l.get_seed()), X)

# Check whether the parameters are different
print(len(params['params']) == 3)
```

```
True
```

```
# tf.keras behaves a bit differently. It removes the duplicate layer
# automatically
shared = tf.keras.layers.Dense(4, activation=tf.nn.relu)
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    shared,
    shared,
    tf.keras.layers.Dense(1),
])

net(X)
# Check whether the parameters are different
print(len(net.layers) == 3)
```

```
True
```

9.2.2 共有パラメータ

しばしば、複数の層にまたがってパラメータを共有したいことがある。それをエレガントに行う方法を見てみよう。以下では全結合層を 1 つ用意し、そのパラメータを使って別の層のパラメータを設定する。ここでは、パラメータにアクセスする前に順伝播 `net(X)` を実行する必要がある。

```
# We need to give the shared layer a name so that we can refer to its
# parameters
shared = nn.LazyLinear(8)
net = nn.Sequential(nn.LazyLinear(8), nn.ReLU(),
                    shared, nn.ReLU(),
                    shared, nn.ReLU(),
                    nn.LazyLinear(1))

net(X)
# Check whether the parameters are the same
print(net[2].weight.data[0] == net[4].weight.data[0])
net[2].weight.data[0, 0] = 100
# Make sure that they are actually the same object rather than just having the
# same value
print(net[2].weight.data[0] == net[4].weight.data[0])
```

```
tensor([True, True, True, True, True, True, True, True])
tensor([True, True, True, True, True, True, True, True])
```

```
net = nn.Sequential()
# We need to give the shared layer a name so that we can refer to its
# parameters
shared = nn.Dense(8, activation='relu')
net.add(nn.Dense(8, activation='relu'),
        shared,
        nn.Dense(8, activation='relu', params=shared.params),
```

(continues on next page)

(continued from previous page)

```
        nn.Dense(10))
net.initialize()

X = np.random.uniform(size=(2, 20))

net(X)
# Check whether the parameters are the same
print(net[1].weight.data()[0] == net[2].weight.data()[0])
net[1].weight.data()[0, 0] = 100
# Make sure that they are actually the same object rather than just having the
# same value
print(net[1].weight.data()[0] == net[2].weight.data()[0])
```

```
[ True  True  True  True  True  True  True  True]
[ True  True  True  True  True  True  True  True]
```

この例は、第 2 層と第 3 層のパラメータが結び付けられていることを示している。それらは単に等しいだけではなく、まったく同じテンソルとして表現されている。したがって、どちらか一方のパラメータを変更すると、もう一方も変化する。

パラメータが共有されているとき、勾配はどうなるのか疑問に思うかもしれない。モデルのパラメータには勾配が含まれているため、第 2 の隠れ層と第 3 の隠れ層の勾配は逆伝播の際に加算される。

9.2.3 まとめ

モデルパラメータにアクセスし、共有するためのいくつかの方法がある。

9.2.4 演習

1. Section 9.1 で定義した `NestMLP` モデルを使い、各層のパラメータにアクセスせよ。
2. 共有パラメータ層を含む MLP を構成して学習せよ。学習の過程で、各層のモデルパラメータと勾配を観察せよ。
3. パラメータ共有はなぜ良い考えなのか。

9.3 パラメータの初期化

パラメータへのアクセス方法がわかったので、次はそれらを適切に初期化する方法を見ていこう。適切な初期化の必要性については Section 8.4 で議論した。深層学習フレームワークは、各層に対してデフォルトのランダム初期化を提供

している。しかし、私たちはしばしば、さまざまな別の手順に従って重みを初期化したいことがある。フレームワークは、最も一般的に使われる手順の多くを提供しており、さらにカスタム初期化子を作成することもできる。

```
import torch
from torch import nn
```

```
from mxnet import init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
import tensorflow as tf
```

デフォルトでは、PyTorch は重み行列とバイアス行列を入力次元と出力次元に基づいて計算される範囲から一様にサンプリングして初期化する。PyTorch の `nn.init` モジュールは、さまざまな既定の初期化方法を提供している。

```
net = nn.Sequential(nn.LazyLinear(8), nn.ReLU(), nn.LazyLinear(1))
X = torch.rand(size=(2, 4))
net(X).shape
```

```
torch.Size([2, 1])
```

```
net = nn.Sequential()
net.add(nn.Dense(8, activation='relu'))
net.add(nn.Dense(1))
net.initialize()  # Use the default initialization method

X = np.random.uniform(size=(2, 4))
net(X).shape
```

```
[07:09:26] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(2, 1)
```

```
net = nn.Sequential([nn.Dense(8), nn.relu, nn.Dense(1)])
X = jax.random.uniform(d2l.get_key(), (2, 4))
params = net.init(d2l.get_key(), X)
net.apply(params, X).shape
```

```
(2, 1)
```

```
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(4, activation=tf.nn.relu),
    tf.keras.layers.Dense(1),
])

X = tf.random.uniform((2, 4))
net(X).shape
```

```
TensorShape([2, 1])
```

9.3.1 組み込み初期化

まずは組み込みの初期化子を呼び出してみよう。以下のコードでは、すべての重みパラメータを標準偏差 0.01 のガウス乱数で初期化し、バイアスパラメータはゼロにする。

```
def init_normal(module):
    if type(module) == nn.Linear:
        nn.init.normal_(module.weight, mean=0, std=0.01)
        nn.init.zeros_(module.bias)

net.apply(init_normal)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([ 0.0082, -0.0003,  0.0037, -0.0037]), tensor(0.))
```

```
# Here force_reinit ensures that parameters are freshly initialized even if
# they were already initialized previously
net.initialize(init=init.Normal(sigma=0.01), force_reinit=True)
net[0].weight.data()[0]
```

```
array([ 0.00354961, -0.00614133,  0.0107317 ,  0.01830765])
```

```
weight_init = nn.initializers.normal(0.01)
bias_init = nn.initializers.zeros
```

(continues on next page)

(continued from previous page)

```
net = nn.Sequential([nn.Dense(8, kernel_init=weight_init, bias_init=bias_init),
                     nn.relu,
                     nn.Dense(1, kernel_init=weight_init, bias_init=bias_
↪init)])

params = net.init(jax.random.PRNGKey(d2l.get_seed()), X)
layer_0 = params['params']['layers_0']
layer_0['kernel'][:, 0], layer_0['bias'][0]
```

```
(Array([-0.00944084,  0.01526781,  0.01000232,  0.01020786], dtype=float32),
 Array(0., dtype=float32))
```

```
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(
        4, activation=tf.nn.relu,
        kernel_initializer=tf.random_normal_initializer(mean=0, stddev=0.01),
        bias_initializer=tf.zeros_initializer()),
    tf.keras.layers.Dense(1)])

net(X)
net.weights[0], net.weights[1]
```

```
(<tf.Variable 'dense_2/kernel:0' shape=(4, 4) dtype=float32, numpy=
 array([[-0.0095617 ,  0.00378142, -0.01553275, -0.00099767],
        [-0.00153092,  0.00172021,  0.01361554, -0.02167145],
        [ 0.00229247, -0.00045055,  0.01760593, -0.00316169],
        [-0.0145327 ,  0.01879665, -0.00351739, -0.003721  ]],
       dtype=float32)>,
 <tf.Variable 'dense_2/bias:0' shape=(4,) dtype=float32, numpy=array([0., 0.,␣
↪0., 0.], dtype=float32)>)
```

すべてのパラメータをある定数値（たとえば 1）に初期化することもできる。

```
def init_constant(module):
    if type(module) == nn.Linear:
        nn.init.constant_(module.weight, 1)
        nn.init.zeros_(module.bias)

net.apply(init_constant)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([1., 1., 1., 1.]), tensor(0.))
```

```
net.initialize(init=init.Constant(1), force_reinit=True)
net[0].weight.data()[0]
```

```
array([1., 1., 1., 1.])
```

```
weight_init = nn.initializers.constant(1)

net = nn.Sequential([nn.Dense(8, kernel_init=weight_init, bias_init=bias_init),
                     nn.relu,
                     nn.Dense(1, kernel_init=weight_init, bias_init=bias_
↪init)])

params = net.init(jax.random.PRNGKey(d2l.get_seed()), X)
layer_0 = params['params']['layers_0']
layer_0['kernel'][:, 0], layer_0['bias'][0]
```

```
(Array([1., 1., 1., 1.], dtype=float32), Array(0., dtype=float32))
```

```
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(
        4, activation=tf.nn.relu,
        kernel_initializer=tf.keras.initializers.Constant(1),
        bias_initializer=tf.zeros_initializer()),
    tf.keras.layers.Dense(1),
])

net(X)
net.weights[0], net.weights[1]
```

```
(<tf.Variable 'dense_4/kernel:0' shape=(4, 4) dtype=float32, numpy=
 array([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]], dtype=float32)>,
 <tf.Variable 'dense_4/bias:0' shape=(4,) dtype=float32, numpy=array([0., 0.,␣
↪0., 0.], dtype=float32)>)
```

特定のブロックに対して異なる初期化子を適用することもできる。たとえば、以下では第 1 層を Xavier 初期化子で初期化し、第 2 層を定数 42 で初期化する。

```
def init_xavier(module):
    if type(module) == nn.Linear:
        nn.init.xavier_uniform_(module.weight)

def init_42(module):
    if type(module) == nn.Linear:
        nn.init.constant_(module.weight, 42)

net[0].apply(init_xavier)
net[2].apply(init_42)
```

(continues on next page)

(continued from previous page)

```
print(net[0].weight.data[0])
print(net[2].weight.data)
```

```
tensor([-0.4081, -0.0622, -0.6532,  0.2202])
tensor([[42., 42., 42., 42., 42., 42., 42., 42.]])
```

```
net[0].weight.initialize(init=init.Xavier(), force_reinit=True)
net[1].initialize(init=init.Constant(42), force_reinit=True)
print(net[0].weight.data()[0])
print(net[1].weight.data())
```

```
[-0.26102373  0.15249556 -0.19274211 -0.24742058]
[[42. 42. 42. 42. 42. 42. 42. 42.]]
```

```
net = nn.Sequential([nn.Dense(8, kernel_init=nn.initializers.xavier_uniform(),
                              bias_init=bias_init),
                     nn.relu,
                     nn.Dense(1, kernel_init=nn.initializers.constant(42),
                              bias_init=bias_init)])

params = net.init(jax.random.PRNGKey(d2l.get_seed()), X)
params['params']['layers_0']['kernel'][:, 0], params['params']['layers_2'][
↪'kernel']
```

```
(Array([-0.3472612 , -0.14502545,  0.5215495 , -0.6690141 ], dtype=float32),
 Array([[42.],
        [42.],
        [42.],
        [42.],
        [42.],
        [42.],
        [42.],
        [42.]], dtype=float32))
```

```
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(
        4,
        activation=tf.nn.relu,
        kernel_initializer=tf.keras.initializers.GlorotUniform()),
    tf.keras.layers.Dense(
        1, kernel_initializer=tf.keras.initializers.Constant(42)),
])

net(X)
print(net.layers[1].weights[0])
print(net.layers[2].weights[0])
```

```
<tf.Variable 'dense_6/kernel:0' shape=(4, 4) dtype=float32, numpy=
array([[ 0.62192386,  0.567151  , -0.83411425,  0.07775843],
       [-0.02987593,  0.69998187, -0.6425164 ,  0.07918394],
       [-0.81396955,  0.48030943, -0.45407206,  0.56459075],
       [ 0.5427986 , -0.27270615, -0.62875164, -0.8139801 ]],
      dtype=float32)>
<tf.Variable 'dense_7/kernel:0' shape=(4, 1) dtype=float32, numpy=
array([[42.],
       [42.],
       [42.],
       [42.]], dtype=float32)>
```

カスタム初期化

ときには、必要な初期化方法が深層学習フレームワークに用意されていないことがある。以下の例では、任意の重みパラメータ w に対して、次の奇妙な分布を用いる初期化子を定義する。

$$w \sim \begin{cases} U(5, 10) & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ U(-10, -5) & \text{with probability } \frac{1}{4} \end{cases} \tag{9.3.1}$$

ここでも、net に適用する my_init 関数を実装する。

```
def my_init(module):
    if type(module) == nn.Linear:
        print("Init", *[(name, param.shape)
                        for name, param in module.named_parameters()][0])
        nn.init.uniform_(module.weight, -10, 10)
        module.weight.data *= module.weight.data.abs() >= 5

net.apply(my_init)
net[0].weight[:2]
```

```
Init weight torch.Size([8, 4])
Init weight torch.Size([1, 8])
```

```
tensor([[ 0.0000,  8.1632,  7.1743, -0.0000],
        [-7.6266,  9.0984, -0.0000,  5.0482]], grad_fn=<SliceBackward0>)
```

```
class MyInit(init.Initializer):
    def _init_weight(self, name, data):
        print('Init', name, data.shape)
        data[:] = np.random.uniform(-10, 10, data.shape)
        data *= np.abs(data) >= 5

net.initialize(MyInit(), force_reinit=True)
net[0].weight.data()[:2]
```

```
Init dense0_weight (8, 4)
Init dense1_weight (1, 8)
```

```
array([[-6.0683527,  8.991421 , -0.       ,  0.       ],
       [ 6.4198647, -9.728567 , -8.057975 ,  0.       ]])
```

```
def my_init(key, shape, dtype=jnp.float_):
    data = jax.random.uniform(key, shape, minval=-10, maxval=10)
    return data * (jnp.abs(data) >= 5)

net = nn.Sequential([nn.Dense(8, kernel_init=my_init), nn.relu, nn.Dense(1)])
params = net.init(d2l.get_key(), X)
print(params['params']['layers_0']['kernel'][:, :2])
```

```
[[ 0.        -9.883148 ]
 [ 9.662153  -8.952053 ]
 [-8.446951   8.958147 ]
 [ 0.         6.0564566]]
```

```
class MyInit(tf.keras.initializers.Initializer):
    def __call__(self, shape, dtype=None):
        data=tf.random.uniform(shape, -10, 10, dtype=dtype)
        factor=(tf.abs(data) >= 5)
        factor=tf.cast(factor, tf.float32)
        return data * factor

net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(
        4,
        activation=tf.nn.relu,
        kernel_initializer=MyInit()),
    tf.keras.layers.Dense(1),
])

net(X)
print(net.layers[1].weights[0])
```

```
<tf.Variable 'dense_8/kernel:0' shape=(4, 4) dtype=float32, numpy=
array([[-9.507499 ,  0.       , -7.383604 ,  7.671566 ],
       [ 8.4090805, -7.128086 ,  0.       ,  0.       ],
       [ 0.       ,  7.5970764,  0.       ,  0.       ],
       [-5.3449297,  6.064953 , -6.477902 ,  0.       ]], dtype=float32)>
```

パラメータを直接設定するという選択肢も常にあることに注意してほしい。

```
net[0].weight.data[:] += 1
```

(continues on next page)

(continued from previous page)

```
net[0].weight.data[0, 0] = 42
net[0].weight.data[0]
```

```
tensor([42.0000,  9.1632,  8.1743,  1.0000])
```

```
net[0].weight.data()[:] += 1
net[0].weight.data()[0, 0] = 42
net[0].weight.data()[0]
```

```
array([42.      ,  9.991421,  1.      ,  1.      ])
```

```
net.layers[1].weights[0][:].assign(net.layers[1].weights[0] + 1)
net.layers[1].weights[0][0, 0].assign(42)
net.layers[1].weights[0]
```

```
<tf.Variable 'dense_8/kernel:0' shape=(4, 4) dtype=float32, numpy=
array([[42.       ,  1.       , -6.383604 ,  8.671566 ],
       [ 9.4090805, -6.128086 ,  1.       ,  1.       ],
       [ 1.       ,  8.597076 ,  1.       ,  1.       ],
       [-4.3449297,  7.064953 , -5.477902 ,  1.       ]], dtype=float32)>
```

9.3.2 まとめ

組み込み初期化子とカスタム初期化子を使ってパラメータを初期化できる。

9.3.3 演習

さらに多くの組み込み初期化子についてオンラインドキュメントを調べてみよう。

9.4 遅延初期化

ここまでのところ、ネットワークの構築においてかなり大雑把にやってもうまくいっているように見えたかもしれない。具体的には、次のような直感に反することを行ってきた。これらは本来うまく動くようには思えないかもしれない。

- 入力次元を指定せずにネットワークアーキテクチャを定義した。
- 直前の層の出力次元を指定せずに層を追加した。

- さらに、モデルが何個のパラメータを持つべきかを決めるのに十分な情報を与える前に、これらのパラメータを「初期化」した。

コードが実際に動いていることに驚くかもしれない。そもそも、深層学習フレームワークがネットワークの入力次元を知る方法はない。ここでの工夫は、フレームワークが初期化を遅延し、最初にデータをモデルに通すまで待って、その場で各層のサイズを推論することである。

後で畳み込みニューラルネットワークを扱うときには、この手法はさらに便利になる。なぜなら、入力次元（たとえば画像の解像度）が、その後に続く各層の次元に影響するからである。したがって、コードを書く時点では次元の値を知らなくてもパラメータを設定できる能力は、モデルの指定やその後の修正を大幅に簡単にしてくれる。それでは、初期化の仕組みをさらに詳しく見ていこう。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
import tensorflow as tf
```

まず、MLP をインスタンス化してみよう。

```
net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))
```

```
net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
```

```
net = nn.Sequential([nn.Dense(256), nn.relu, nn.Dense(10)])
```

```
net = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    tf.keras.layers.Dense(10),
])
```

この時点では、入力次元がまだ不明なので、ネットワークは入力層の重みの次元を知ることができない。

したがって、フレームワークはまだどのパラメータも初期化していない。以下でパラメータにアクセスしようとして確認してみよう。

```
net[0].weight
```

```
<UninitializedParameter>
```

```
print(net.collect_params)
print(net.collect_params())
```

```
<bound method Block.collect_params of Sequential(
  (0): Dense(-1 -> 256, Activation(relu))
  (1): Dense(-1 -> 10, linear)
)>
sequential0_ (
  Parameter dense0_weight (shape=(256, -1), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, -1), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

```
params = net.init(d2l.get_key(), jnp.zeros((2, 20)))
jax.tree_util.tree_map(lambda x: x.shape, params).tree_flatten_with_keys()
```

```
(((DictKey(key='params'),
   {'layers_0': {'bias': (256,), 'kernel': (20, 256)},
    'layers_2': {'bias': (10,), 'kernel': (256, 10)}}),),
 ('params',))
```

```
[net.layers[i].get_weights() for i in range(len(net.layers))]
```

```
[[], []]
```

次に、ネットワークにデータを通して、フレームワークにようやくパラメータを初期化させてみよう。

```
X = torch.rand(2, 20)
net(X)

net[0].weight.shape
```

```
torch.Size([256, 20])
```

```
net.initialize()
net.collect_params()
```

```
[07:07:50] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, -1), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, -1), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

```
@d2l.add_to_class(d2l.Module)  #@save
def apply_init(self, dummy_input, key):
    params = self.init(key, *dummy_input)  # dummy_input tuple unpacked
    return params
```

```
X = tf.random.uniform((2, 20))
net(X)
[w.shape for w in net.get_weights()]
```

```
[(20, 256), (256,), (256, 10), (10,)]
```

入力次元 20 が分かればすぐに、フレームワークは 20 の値を代入することで最初の層の重み行列の形状を特定できる。最初の層の形状が分かると、フレームワークは次の層へ進み、計算グラフに沿って順に処理し、すべての形状が分かるまで続ける。この場合、遅延初期化が必要なのは最初の層だけだが、フレームワークは順次に初期化を行う。すべてのパラメータ形状が分かると、フレームワークはようやくパラメータを初期化できる。

次のメソッドは、ダミー入力をネットワークに通して予備実行を行い、すべてのパラメータ形状を推論したうえでパラメータを初期化する。これは、デフォルトのランダム初期化を望まない場合に後で使われる。

```
@d2l.add_to_class(d2l.Module)  #@save
def apply_init(self, inputs, init=None):
```

(continues on next page)

(continued from previous page)

```
    self.forward(*inputs)
    if init is not None:
        self.net.apply(init)
```

```
X = np.random.uniform(size=(2, 20))
net(X)

net.collect_params()
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

9.4.1 要約

遅延初期化は便利である。フレームワークがパラメータ形状を自動的に推論できるため、アーキテクチャの修正が容易になり、よくあるエラーの原因を一つ取り除ける。モデルにデータを通すことで、フレームワークにようやくパラメータを初期化させることができる。

9.4.2 演習

1. 最初の層には入力次元を指定するが、その後の層には指定しない場合、どうなるか？すぐに初期化されるか？
2. 次元が一致しないように指定した場合、どうなるか？
3. 入力の次元が変化する場合、何をする必要があるか？ヒント: パラメータ共有を見てみよう。

9.5 カスタム層

ディープラーニングの成功を支える要因の一つは、さまざまな層が利用できることである。それらを創造的に組み合わせることで、多種多様なタスクに適したアーキテクチャを設計できる。たとえば研究者たちは、画像、テキスト、系列データの反復処理、そして動的計画法の実行に特化した層を発明してきた。遅かれ早かれ、ディープラーニングフレームワークにはまだ存在しない層が必要になるだろう。そのような場合には、カスタム層を作成しなければならない。この節では、その方法を示す。

```
from d2l import torch as d2l
import torch
from torch import nn
from torch.nn import functional as F
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

9.5.1 パラメータを持たない層

まず、独自のパラメータを一切持たないカスタム層を構築してみよう。これは、Section 9.1 でモジュールを導入したときの内容を覚えていれば、見覚えがあるはずである。次の CenteredLayer クラスは、入力から単に平均を引くだけである。これを作るには、基底層クラスを継承し、順伝播関数を実装するだけで十分である。

```
class CenteredLayer(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return X - X.mean()
```

```
class CenteredLayer(nn.Block):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def forward(self, X):
        return X - X.mean()
```

```
class CenteredLayer(nn.Module):
    def __call__(self, X):
        return X - X.mean()
```

```
class CenteredLayer(tf.keras.Model):
    def __init__(self):
        super().__init__()

    def call(self, X):
        return X - tf.reduce_mean(X)
```

いくつかのデータをこの層に通して、意図したとおりに動作することを確認しよう。

```
layer = CenteredLayer()
layer(d2l.tensor([1.0, 2, 3, 4, 5]))
```

```
tensor([-2., -1.,  0.,  1.,  2.])
```

これで、この層をより複雑なモデルを構築する際の構成要素として組み込める。

```
net = nn.Sequential(nn.LazyLinear(128), CenteredLayer())
```

```
net = nn.Sequential()
net.add(nn.Dense(128), CenteredLayer())
net.initialize()
```

```
net = nn.Sequential([nn.Dense(128), CenteredLayer()])
```

```
net = tf.keras.Sequential([tf.keras.layers.Dense(128), CenteredLayer()])
```

追加の健全性チェックとして、ランダムなデータをネットワークに通し、平均が実際に 0 であることを確認できる。浮動小数点数を扱っているため、量子化の影響でごく小さな 0 でない値が見えることもある。

```
Y = net(d2l.rand(4, 8))
Y.mean()
```

```
tensor(-8.3819e-09, grad_fn=<MeanBackward0>)
```

```
Y = net(d2l.rand(4, 8))
Y.mean()
```

```
array(3.783498e-10)
```

```
Y, _ = net.init_with_output(d2l.get_key(), jax.random.uniform(d2l.get_key(),
                                                              (4, 8)))
Y.mean()
```

```
Array(5.5879354e-09, dtype=float32)
```

```
Y = net(tf.random.uniform((4, 8)))
tf.reduce_mean(Y)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=4.1909516e-09>
```

9.5.2 パラメータを持つ層

単純な層の定義方法がわかったので、次は、学習によって調整できるパラメータを持つ層の定義に進もう。組み込み関数を使ってパラメータを作成でき、それらは基本的な管理機能を提供する。特に、アクセス、初期化、共有、保存、モデルパラメータの読み込みを管理する。このようにして、ほかにも利点はあるが、カスタム層ごとに独自のシリアライズ処理を書く必要がなくなる。

では、全結合層の独自実装を行ってみよう。この層には 2 つのパラメータが必要であることを思い出してほしい。一つは重みを表し、もう一つはバイアスである。この実装では、ReLU 活性化をデフォルトとして組み込んでいる。この層は `in_units` と `units` の 2 つの入力引数を必要とし、それぞれ入力数と出力数を表す。

```
class MyLinear(nn.Module):
    def __init__(self, in_units, units):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(in_units, units))
        self.bias = nn.Parameter(torch.randn(units,))

    def forward(self, X):
        linear = torch.matmul(X, self.weight.data) + self.bias.data
        return F.relu(linear)
```

```
class MyDense(nn.Block):
    def __init__(self, units, in_units, **kwargs):
        super().__init__(**kwargs)
        self.weight = self.params.get('weight', shape=(in_units, units))
        self.bias = self.params.get('bias', shape=(units,))

    def forward(self, x):
        linear = np.dot(x, self.weight.data(ctx=x.ctx)) + self.bias.data(
            ctx=x.ctx)
        return npx.relu(linear)
```

```
class MyDense(nn.Module):
    in_units: int
    units: int

    def setup(self):
        self.weight = self.param('weight', nn.initializers.normal(stddev=1),
                                 (self.in_units, self.units))
        self.bias = self.param('bias', nn.initializers.zeros, self.units)

    def __call__(self, X):
        linear = jnp.matmul(X, self.weight) + self.bias
        return nn.relu(linear)
```

```
class MyDense(tf.keras.Model):
    def __init__(self, units):
        super().__init__()
        self.units = units

    def build(self, X_shape):
        self.weight = self.add_weight(name='weight',
            shape=[X_shape[-1], self.units],
            initializer=tf.random_normal_initializer())
        self.bias = self.add_weight(
            name='bias', shape=[self.units],
            initializer=tf.zeros_initializer())

    def call(self, X):
        linear = tf.matmul(X, self.weight) + self.bias
        return tf.nn.relu(linear)
```

次に、MyDense クラスをインスタンス化し、そのモデルパラメータにアクセスする。

次に、MyLinear クラスをインスタンス化し、そのモデルパラメータにアクセスする。

```
linear = MyLinear(5, 3)
linear.weight
```

```
Parameter containing:
tensor([[-0.5519, -0.3220,  1.2418],
        [ 0.3559,  2.2357, -0.5870],
        [ 1.1292, -0.0048, -0.7610],
        [-1.5225, -0.2633,  0.1360],
        [-1.8041, -0.6308, -0.0188]], requires_grad=True)
```

```
dense = MyDense(units=3, in_units=5)
dense.params
```

```
mydense0_ (
  Parameter mydense0_weight (shape=(5, 3), dtype=<class 'numpy.float32'>)
  Parameter mydense0_bias (shape=(3,), dtype=<class 'numpy.float32'>)
)
```

```
dense = MyDense(5, 3)
params = dense.init(d2l.get_key(), jnp.zeros((3, 5)))
params
```

```
FrozenDict({
    params: {
        weight: Array([[ 0.457323  ,  0.48909545, -0.69718534],
               [ 0.39462337, -1.9821982 ,  0.26406184],
               [-1.2466857 , -0.6145607 , -0.48282492],
               [-0.25153905,  0.48479155,  0.35480422],
               [-1.6592307 ,  0.8844818 ,  1.2634686 ]], dtype=float32),
        bias: Array([0., 0., 0.], dtype=float32),
    },
})
```

```
dense = MyDense(3)
dense(tf.random.uniform((2, 5)))
dense.get_weights()
```

```
[array([[ 0.05366681,  0.10157764, -0.04425018],
        [-0.00799372, -0.10843283,  0.02491164],
        [-0.00250257,  0.04952564,  0.0640744 ],
        [-0.05546394, -0.00562379, -0.03835879],
        [ 0.09050553,  0.06934955,  0.02847055]], dtype=float32),
 array([0., 0., 0.], dtype=float32)]
```

カスタム層を使って、順伝播計算を直接実行することもできる。

```
linear(torch.rand(2, 5))
```

```
tensor([[0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.1040]])
```

```
dense.initialize()
dense(np.random.uniform(size=(2, 5)))
```

```
array([[0.        , 0.01633355, 0.        ],
       [0.        , 0.01581812, 0.        ]])
```

```
dense.apply(params, jax.random.uniform(d2l.get_key(),
                                       (2, 5)))
```

```
Array([[0.       , 0.       , 0.       ],
       [0.       , 0.       , 0.6575615]], dtype=float32)
```

```
dense(tf.random.uniform((2, 5)))
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0.        , 0.        , 0.        ],
       [0.12537432, 0.13187361, 0.02326545]], dtype=float32)>
```

また、カスタム層を使ってモデルを構築することもできる。一度それができれば、組み込みの全結合層と同じように使える。

```
net = nn.Sequential(MyLinear(64, 8), MyLinear(8, 1))
net(torch.rand(2, 64))
```

```
tensor([[0.],
        [0.]])
```

```
net = nn.Sequential()
net.add(MyDense(8, in_units=64),
        MyDense(1, in_units=8))
net.initialize()
net(np.random.uniform(size=(2, 64)))
```

```
array([[0.06508517],
       [0.0615553 ]])
```

```
net = nn.Sequential([MyDense(64, 8), MyDense(8, 1)])
Y, _ = net.init_with_output(d2l.get_key(), jax.random.uniform(d2l.get_key(),
                                                              (2, 64)))
Y
```

```
Array([[5.096267],
       [4.02948 ]], dtype=float32)
```

```
net = tf.keras.models.Sequential([MyDense(8), MyDense(1)])
net(tf.random.uniform((2, 64)))
```

```
<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
array([[0.01015611],
       [0.01611123]], dtype=float32)>
```

9.5.3 まとめ

基本的な層クラスを使ってカスタム層を設計できる。これにより、ライブラリに既存のどの層とも異なる、柔軟な新しい層を定義できる。一度定義すれば、カスタム層は任意の文脈やアーキテクチャで呼び出せる。層はローカルなパラメータを持つことができ、それらは組み込み関数を通じて作成できる。

9.5.4 演習

1. 入力を受け取りテンソル縮約を計算する層を設計せよ。つまり、$y_k = \sum_{i,j} W_{ijk} x_i x_j$ を返す層である。
2. データのフーリエ係数の前半を返す層を設計せよ。

9.6 ファイル入出力

これまで、データの処理方法や、深層学習モデルの構築、学習、テストの方法について議論してきた。しかし、いずれは学習済みモデルに十分満足し、さまざまな文脈で後から利用するために結果を保存したくなるはずである（あるいは、デプロイ時に予測を行うためかもしれない）。さらに、長時間の学習処理を実行しているときには、途中結果を定期的に保存する（チェックポイントを取る）ことがベストプラクティスである。そうしておけば、サーバーの電源コードにつまずいてしまっても、数日分の計算を失わずに済む。そこで、個々の重みベクトルとモデル全体の両方を、どのように読み書きするかを学ぶ時が来た。この節ではその両方を扱う。

```
import torch
from torch import nn
from torch.nn import functional as F
```

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
import flax
from flax import linen as nn
from flax.training import checkpoints
```

(continues on next page)

(continued from previous page)

```
import jax
from jax import numpy as jnp
```

```
WARNING:absl:GlobalAsyncCheckpointManager is not imported correctly.␣
↪Checkpointing of GlobalDeviceArrays will not be available.To use the feature,
↪ install tensorstore.
WARNING:jax._src.xla_bridge:No GPU/TPU found, falling back to CPU. (Set TF_CPP_
↪MIN_LOG_LEVEL=0 and rerun for more info.)
```

```
import tensorflow as tf
import numpy as np
```

9.6.1 テンソルの読み込みと保存

個々のテンソルについては、load と save 関数を直接呼び出して、それぞれ読み込みと書き込みを行える。どちらの関数も名前を指定する必要があり、save には保存する変数を入力として与える必要がある。

```
x = torch.arange(4)
torch.save(x, 'x-file')
```

```
x = np.arange(4)
npx.save('x-file', x)
```

```
[07:15:22] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
x = jnp.arange(4)
jnp.save('x-file.npy', x)
```

```
x = tf.range(4)
np.save('x-file.npy', x)
```

これで、保存されたファイルからデータをメモリに読み戻せる。

```
x2 = torch.load('x-file')
x2
```

```
tensor([0, 1, 2, 3])
```

```
x2 = npx.load('x-file')
x2
```

```
[array([0., 1., 2., 3.])]
```

```
x2 = jnp.load('x-file.npy', allow_pickle=True)
x2
```

```
Array([0, 1, 2, 3], dtype=int32)
```

```
x2 = np.load('x-file.npy', allow_pickle=True)
x2
```

```
array([0, 1, 2, 3], dtype=int32)
```

テンソルのリストを保存して、メモリに読み戻すこともできる。

```
y = torch.zeros(4)
torch.save([x, y],'x-files')
x2, y2 = torch.load('x-files')
(x2, y2)
```

```
(tensor([0, 1, 2, 3]), tensor([0., 0., 0., 0.]))
```

```
y = np.zeros(4)
npx.save('x-files', [x, y])
x2, y2 = npx.load('x-files')
(x2, y2)
```

```
(array([0., 1., 2., 3.]), array([0., 0., 0., 0.]))
```

```
y = jnp.zeros(4)
jnp.save('xy-files.npy', [x, y])
x2, y2 = jnp.load('xy-files.npy', allow_pickle=True)
(x2, y2)
```

```
(Array([0., 1., 2., 3.], dtype=float32),
 Array([0., 0., 0., 0.], dtype=float32))
```

```
y = tf.zeros(4)
np.save('xy-files.npy', [x, y])
x2, y2 = np.load('xy-files.npy', allow_pickle=True)
(x2, y2)
```

```
(array([0., 1., 2., 3.]), array([0., 0., 0., 0.]))
```

さらに、文字列からテンソルへの対応を表す辞書を書き込んで読み込むこともできる。これは、モデル内のすべての重みを読み書きしたいときに便利である。

```
mydict = {'x': x, 'y': y}
torch.save(mydict, 'mydict')
mydict2 = torch.load('mydict')
mydict2
```

```
{'x': tensor([0, 1, 2, 3]), 'y': tensor([0., 0., 0., 0.])}
```

```
mydict = {'x': x, 'y': y}
npx.save('mydict', mydict)
mydict2 = npx.load('mydict')
mydict2
```

```
{'x': array([0., 1., 2., 3.]), 'y': array([0., 0., 0., 0.])}
```

```
mydict = {'x': x, 'y': y}
jnp.save('mydict.npy', mydict)
mydict2 = jnp.load('mydict.npy', allow_pickle=True)
mydict2
```

```
array({'x': Array([0, 1, 2, 3], dtype=int32), 'y': Array([0., 0., 0., 0.],␣
↪dtype=float32)},
      dtype=object)
```

```
mydict = {'x': x, 'y': y}
np.save('mydict.npy', mydict)
mydict2 = np.load('mydict.npy', allow_pickle=True)
mydict2
```

```
array({'x': <tf.Tensor: shape=(4,), dtype=int32, numpy=array([0, 1, 2, 3],␣
↪dtype=int32)>, 'y': <tf.Tensor: shape=(4,), dtype=float32, numpy=array([0.,␣
↪0., 0., 0.], dtype=float32)>},
      dtype=object)
```

9.6.2 モデルパラメータの読み込みと保存

個々の重みベクトル（あるいは他のテンソル）を保存するのは便利だが、モデル全体を保存（および後で読み込み）したい場合には非常に面倒になる。何しろ、あちこちに散らばった多数のパラメータ群があるかもしれないからであ

る。このため、深層学習フレームワークには、ネットワーク全体を読み込み・保存するための組み込み機能が用意されている。ここで注意すべき重要な点は、これが保存するのはモデルのパラメータであって、モデル全体ではないということである。たとえば、3 層の MLP がある場合、アーキテクチャは別途指定する必要がある。その理由は、モデル自体が任意のコードを含みうるため、自然な形ではシリアライズできないからである。したがって、モデルを復元するには、コードでアーキテクチャを生成し、その後でディスクからパラメータを読み込む必要がある。まずはおなじみの MLP から始めよう。

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.LazyLinear(256)
        self.output = nn.LazyLinear(10)

    def forward(self, x):
        return self.output(F.relu(self.hidden(x)))

net = MLP()
X = torch.randn(size=(2, 20))
Y = net(X)
```

```
class MLP(nn.Block):
    def __init__(self, **kwargs):
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu')
        self.output = nn.Dense(10)

    def forward(self, x):
        return self.output(self.hidden(x))

net = MLP()
net.initialize()
X = np.random.uniform(size=(2, 20))
Y = net(X)
```

```
class MLP(nn.Module):
    def setup(self):
        self.hidden = nn.Dense(256)
        self.output = nn.Dense(10)

    def __call__(self, x):
        return self.output(nn.relu(self.hidden(x)))

net = MLP()
X = jax.random.normal(jax.random.PRNGKey(d2l.get_seed()), (2, 20))
Y, params = net.init_with_output(jax.random.PRNGKey(d2l.get_seed()), X)
```

```
class MLP(tf.keras.Model):
    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        self.flatten = tf.keras.layers.Flatten()
        self.hidden = tf.keras.layers.Dense(units=256, activation=tf.nn.relu)
        self.out = tf.keras.layers.Dense(units=10)

    def call(self, inputs):
        x = self.flatten(inputs)
        x = self.hidden(x)
        return self.out(x)

net = MLP()
X = tf.random.uniform((2, 20))
Y = net(X)
```

次に、モデルのパラメータを“mlp.params”という名前のファイルとして保存する。

```
torch.save(net.state_dict(), 'mlp.params')
```

```
net.save_parameters('mlp.params')
```

```
checkpoints.save_checkpoint('ckpt_dir', params, step=1, overwrite=True)
```

```
'ckpt_dir/checkpoint_1'
```

```
net.save_weights('mlp.params')
```

モデルを復元するには、元の MLP モデルのクローンをインスタンス化する。モデルパラメータをランダムに初期化する代わりに、ファイルに保存されたパラメータを直接読み込む。

```
clone = MLP()
clone.load_state_dict(torch.load('mlp.params'))
clone.eval()
```

```
MLP(
  (hidden): LazyLinear(in_features=0, out_features=256, bias=True)
  (output): LazyLinear(in_features=0, out_features=10, bias=True)
)
```

```
clone = MLP()
clone.load_parameters('mlp.params')
```

```
clone = MLP()
```

(continues on next page)

(continued from previous page)

```
cloned_params = flax.core.freeze(checkpoints.restore_checkpoint('ckpt_dir',
                                                                target=None))
```

```
clone = MLP()
clone.load_weights('mlp.params')
```

```
<tensorflow.python.checkpoint.checkpoint.CheckpointLoadStatus at␣
↪0x7f912199d1b0>
```

両方のインスタンスは同じモデルパラメータを持っているので、同じ入力 `X` に対する計算結果も同じになるはずである。確認してみよう。

```
Y_clone = clone(X)
Y_clone == Y
```

```
tensor([[True, True, True, True, True, True, True, True, True, True],
        [True, True, True, True, True, True, True, True, True, True]])
```

```
Y_clone = clone(X)
Y_clone == Y
```

```
array([[ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True]])
```

```
Y_clone = clone.apply(cloned_params, X)
Y_clone == Y
```

```
Array([[ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True]], dtype=bool)
```

```
Y_clone = clone(X)
Y_clone == Y
```

```
<tf.Tensor: shape=(2, 10), dtype=bool, numpy=
array([[ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True]])>
```

9.6.3 まとめ

save と load 関数は、テンソルオブジェクトに対するファイル入出力に使える。ネットワーク全体のパラメータ集合は、パラメータ辞書を介して保存・読み込みできる。アーキテクチャの保存は、パラメータではなくコードで行う必要がある。

9.6.4 演習

1. 学習済みモデルを別のデバイスにデプロイする必要がない場合でも、モデルパラメータを保存する実用上の利点は何か？
2. 異なるアーキテクチャを持つネットワークに組み込むために、ネットワークの一部だけを再利用したいとする。たとえば、以前のネットワークの最初の2層を新しいネットワークで使うには、どうすればよいだろうか？
3. ネットワークのアーキテクチャとパラメータを保存するには、どうすればよいだろうか？アーキテクチャにはどのような制約を課すか？

9.7 GPU

tab_intro_decade では、過去 20 年間の計算能力の急速な成長を示した。要するに、2000 年以降、GPU の性能は 10 年ごとに 1000 倍に向上してきた。これは大きな機会をもたらすが、同時に、そのような性能に対する大きな需要があったことも示している。

この節では、この計算性能を研究にどう活用するかを説明し始める。まずは単一 GPU の使い方を取り上げ、後の段階で複数 GPU や複数サーバー（複数 GPU 搭載）を使う方法を扱う。

具体的には、単一の NVIDIA GPU を計算に使う方法を説明する。まず、少なくとも 1 枚の NVIDIA GPU が搭載されていることを確認してほしい。次に、NVIDIA driver と CUDA[74] をダウンロードし、指示に従って適切なパスを設定する。これらの準備が完了したら、nvidia-smi コマンドを使ってグラフィックカード情報を表示できる。



PyTorch では、すべての配列にデバイスがあり、これを コンテキストと呼ぶことがよくある。これまでのところ、デフォルトでは、すべての変数と関連する計算は CPU に割り当てられていた。通常、他のコンテキストとしてはさまざまな GPU がある。複数のサーバーにまたがってジョブを展開すると、状況はさらに複雑になる。配列をコンテキストに賢く割り当てることで、デバイス間でデータを転送する時間を最小化できる。たとえば、GPU 搭載サーバーでニューラルネットワークを学習するときには、モデルのパラメータを GPU 上に置くのが普通である。

この節のプログラムを実行するには、少なくとも 2 枚の GPU が必要である。これは多くのデスクトップコンピュータにとっては贅沢かもしれないが、たとえば AWS EC2 のマルチ GPU インスタンスを使えば、クラウド上では簡単に利用できる。他のほとんどの節では複数 GPU は 不要であるが、ここでは単に異なるデバイス間でのデータの流れを示したいだけである。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

9.7.1 計算デバイス

CPU や GPU などのデバイスを、保存や計算のために指定できる。デフォルトでは、テンソルは主記憶に作成され、その後 CPU が計算に使われる。

PyTorch では、CPU と GPU は `torch.device('cpu')` と `torch.device('cuda')` で指定できる。cpu デバイスは、すべての物理 CPU とメモリを意味することに注意してほしい。これは、PyTorch の計算がすべての CPU コアを使おうとすることを意味する。一方、gpu デバイスは 1 枚のカードとそれに対応するメモリだけを表す。複数の GPU がある場合は、i^{th} GPU（i は 0 から始まる）を表すために `torch.device(f'cuda:{i}')` を使う。また、`gpu:0` と gpu は等価である。

```
def cpu():  #@save
    """Get the CPU device."""
    return torch.device('cpu')

def gpu(i=0):  #@save
    """Get a GPU device."""
    return torch.device(f'cuda:{i}')

cpu(), gpu(), gpu(1)
```

```
(device(type='cpu'),
```

(continues on next page)

(continued from previous page)

```
 device(type='cuda', index=0),
 device(type='cuda', index=1))
```

```
def cpu():  #@save
    """Get the CPU device."""
    if tab.selected('mxnet'):
        return npx.cpu()
    if tab.selected('tensorflow'):
        return tf.device('/CPU:0')
    if tab.selected('jax'):
        return jax.devices('cpu')[0]

def gpu(i=0):  #@save
    """Get a GPU device."""
    if tab.selected('mxnet'):
        return npx.gpu(i)
    if tab.selected('tensorflow'):
        return tf.device(f'/GPU:{i}')
    if tab.selected('jax'):
        return jax.devices('gpu')[i]

cpu(), gpu(), gpu(1)
```

```
(cpu(0), gpu(0), gpu(1))
```

```
def cpu():  #@save
    """Get the CPU device."""
    if tab.selected('mxnet'):
        return npx.cpu()
    if tab.selected('tensorflow'):
        return tf.device('/CPU:0')
    if tab.selected('jax'):
        return jax.devices('cpu')[0]

def gpu(i=0):  #@save
    """Get a GPU device."""
    if tab.selected('mxnet'):
        return npx.gpu(i)
    if tab.selected('tensorflow'):
        return tf.device(f'/GPU:{i}')
    if tab.selected('jax'):
        return jax.devices('gpu')[i]

cpu(), gpu(), gpu(1)
```

```
(CpuDevice(id=0), gpu(id=0), gpu(id=1))
```

```
def cpu():  #@save
    """Get the CPU device."""
```

(continues on next page)

(continued from previous page)

```
    if tab.selected('mxnet'):
        return npx.cpu()
    if tab.selected('tensorflow'):
        return tf.device('/CPU:0')
    if tab.selected('jax'):
        return jax.devices('cpu')[0]

def gpu(i=0):  #@save
    """Get a GPU device."""
    if tab.selected('mxnet'):
        return npx.gpu(i)
    if tab.selected('tensorflow'):
        return tf.device(f'/GPU:{i}')
    if tab.selected('jax'):
        return jax.devices('gpu')[i]

cpu(), gpu(), gpu(1)
```

```
(<tensorflow.python.eager.context._EagerDeviceContext at 0x7f58eef46000>,
 <tensorflow.python.eager.context._EagerDeviceContext at 0x7f58eddb3d40>,
 <tensorflow.python.eager.context._EagerDeviceContext at 0x7f58edc23980>)
```

利用可能な GPU の数を問い合わせることができる。

```
def num_gpus():  #@save
    """Get the number of available GPUs."""
    return torch.cuda.device_count()

num_gpus()
```

```
2
```

```
def num_gpus():  #@save
    """Get the number of available GPUs."""
    if tab.selected('mxnet'):
        return npx.num_gpus()
    if tab.selected('tensorflow'):
        return len(tf.config.experimental.list_physical_devices('GPU'))
    if tab.selected('jax'):
        try:
            return jax.device_count('gpu')
        except:
            return 0  # No GPU backend found

num_gpus()
```

```
2
```

```
def num_gpus():  #@save
    """Get the number of available GPUs."""
    if tab.selected('mxnet'):
        return npx.num_gpus()
    if tab.selected('tensorflow'):
        return len(tf.config.experimental.list_physical_devices('GPU'))
    if tab.selected('jax'):
        try:
            return jax.device_count('gpu')
        except:
            return 0  # No GPU backend found

num_gpus()
```

```
2
```

```
def num_gpus():  #@save
    """Get the number of available GPUs."""
    if tab.selected('mxnet'):
        return npx.num_gpus()
    if tab.selected('tensorflow'):
        return len(tf.config.experimental.list_physical_devices('GPU'))
    if tab.selected('jax'):
        try:
            return jax.device_count('gpu')
        except:
            return 0  # No GPU backend found

num_gpus()
```

```
2
```

ここで、要求した GPU が存在しなくてもコードを実行できるようにする、便利な 2 つの関数を定義する。

```
def try_gpu(i=0):  #@save
    """Return gpu(i) if exists, otherwise return cpu()."""
    if num_gpus() >= i + 1:
        return gpu(i)
    return cpu()

def try_all_gpus():  #@save
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    return [gpu(i) for i in range(num_gpus())]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(device(type='cuda', index=0),
 device(type='cpu'),
 [device(type='cuda', index=0), device(type='cuda', index=1)])
```

9.7.2 テンソルと GPU

デフォルトでは、テンソルは CPU 上に作成される。テンソルがどのデバイス上にあるかを問い合わせることができる。

デフォルトでは、利用可能であればテンソルは GPU/TPU 上に作成され、利用できない場合は CPU が使われる。テンソルがどのデバイス上にあるかを問い合わせることができる。

```
x = torch.tensor([1, 2, 3])
x.device
```

```
device(type='cpu')
```

```
x = np.array([1, 2, 3])
x.ctx
```

```
[07:26:55] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
cpu(0)
```

```
x = jnp.array([1, 2, 3])
x.device()
```

```
gpu(id=0)
```

```
x = tf.constant([1, 2, 3])
x.device
```

```
'/job:localhost/replica:0/task:0/device:GPU:0'
```

複数の項を扱うときには、それらが同じデバイス上になければならないことに注意することが重要である。たとえば、2 つのテンソルを足し合わせる場合、両方の引数が同じデバイス上にあることを確認する必要がある。そうでなければ、フレームワークは結果をどこに保存すべきか、あるいは計算をどこで行うべきかさえ判断できない。

GPU 上での保存

テンソルを GPU 上に保存する方法はいくつかある。たとえば、テンソルを作成するときに保存先デバイスを指定できる。次に、最初の gpu 上にテンソル変数 X を作成する。GPU 上で作成されたテンソルは、その GPU のメモリだけを

消費する。nvidia-smi コマンドを使って GPU のメモリ使用量を確認できる。一般に、GPU メモリの制限を超えるデータを作成しないように注意する必要がある。

```
X = torch.ones(2, 3, device=try_gpu())
X
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
```

```
X = np.ones((2, 3), ctx=try_gpu())
X
```

```
[07:26:56] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

```
array([[1., 1., 1.],
       [1., 1., 1.]], ctx=gpu(0))
```

```
# By default JAX puts arrays to GPUs or TPUs if available
X = jax.device_put(jnp.ones((2, 3)), try_gpu())
X
```

```
Array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
```

```
with try_gpu():
    X = tf.ones((2, 3))
X
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)>
```

少なくとも 2 枚の GPU があると仮定すると、次のコードは 2 枚目の GPU 上にランダムなテンソル Y を作成する。

```
Y = torch.rand(2, 3, device=try_gpu(1))
Y
```

```
tensor([[0.6195, 0.6906, 0.3363],
        [0.6355, 0.5201, 1.0000]], device='cuda:1')
```

```
Y = np.random.uniform(size=(2, 3), ctx=try_gpu(1))
Y
```

```
[07:26:56] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

```
array([[0.67478997, 0.07540122, 0.9956977 ],
       [0.09488854, 0.415456  , 0.11231736]], ctx=gpu(1))
```

```
Y = jax.device_put(jax.random.uniform(jax.random.PRNGKey(0), (2, 3)),
                   try_gpu(1))
Y
```

```
Array([[0.57450044, 0.09968603, 0.7419659 ],
       [0.8941783 , 0.59656656, 0.45325184]], dtype=float32)
```

```
with try_gpu(1):
    Y = tf.random.uniform((2, 3))
Y
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0.99093616, 0.51236033, 0.7667209 ],
       [0.618168  , 0.34049547, 0.804111  ]], dtype=float32)>
```

コピー

X + Y を計算したい場合、どこでこの演算を行うかを決める必要がある。たとえば、Fig. 9.7.1 に示すように、X を 2 枚目の GPU に転送してそこで演算を実行できる。単に X と Y を足しては いけない。そうすると例外が発生する。ランタイムエンジンは何をすべきかわからず、同じデバイス上にデータを見つけられないため失敗する。Y は 2 枚目の GPU 上にあるので、2 つを足す前に X をそこへ移動する必要がある。

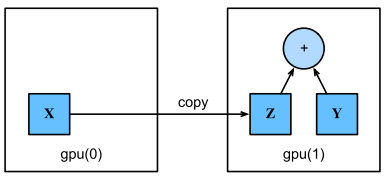


Fig. 9.7.1 Copy data to perform an operation on the same device.

```
Z = X.cuda(1)
print(X)
print(Z)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:1')
```

```
Z = X.copyto(try_gpu(1))
print(X)
print(Z)
```

```
[[1. 1. 1.]
 [1. 1. 1.]] @gpu(0)
[[1. 1. 1.]
 [1. 1. 1.]] @gpu(1)
```

```
Z = jax.device_put(X, try_gpu(1))
print(X)
print(Z)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
[[1. 1. 1.]
 [1. 1. 1.]]
```

```
with try_gpu(1):
    Z = X
print(X)
print(Z)
```

```
tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]], shape=(2, 3), dtype=float32)
tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]], shape=(2, 3), dtype=float32)
```

これでデータ（`Z` と `Y` の両方）が同じ GPU 上にあるので、それらを加算できる。

```
Y + Z
```

```
tensor([[1.6195, 1.6906, 1.3363],
        [1.6355, 1.5201, 2.0000]], device='cuda:1')
```

しかし、変数 `Z` がすでに 2 枚目の GPU 上にあるとしたらどうだろうか？それでも `Z.cuda(1)` を呼び出したらどうなるだろうか？コピーを作成して新しいメモリを割り当てる代わりに、`Z` を返す。

```
Z.cuda(1) is Z
```

```
True
```

```
Z.as_in_ctx(try_gpu(1)) is Z
```

```
True
```

```
Z2 = jax.device_put(Z, try_gpu(1))
Z2 is Z
```

```
False
```

```
with try_gpu(1):
    Z2 = Z
Z2 is Z
```

```
True
```

補足

人々が GPU を機械学習に使うのは、それが高速だと期待しているからである。しかし、変数をデバイス間で転送するのは遅く、計算よりもずっと遅いのである。そのため、何か遅いことをさせる前に、それを本当にやりたいのか 100% 確信してもらう必要がある。深層学習フレームワークが、クラッシュせずに自動でコピーしてしまうだけだと、遅いコードを書いてしまったことに気づかないかもしれない。

データ転送は遅いだけでなく、並列化もずっと難しくする。というのも、次の操作に進む前にデータが送られてくるのを（正確には受け取られるのを）待たなければならないからである。そのため、コピー操作は非常に慎重に扱う必要がある。経験則として、多くの小さな操作は 1 つの大きな操作よりもはるかに悪い。さらに、何をしているかをよく理解しているのでない限り、コードの中に多数の単独の操作を散りばめるよりも、複数の操作をまとめて行うほうがずっと良い。これは、そのような操作が、一方のデバイスが他方を待たなければ次のことができない場合にブロックされうるからである。これは、電話で事前注文しておいて、あなたが来たときにはもうできあがっているコーヒーを受け取るのではなく、列に並んでコーヒーを注文するのに少し似ている。

最後に、テンソルを表示したり NumPy 形式に変換したりするとき、データが主記憶にない場合、フレームワークはまずそれを主記憶にコピーするため、追加の転送オーバーヘッドが発生する。さらに悪いことに、そのデータは Python の

完了をすべて待たせる悪名高いグローバルインタプリタロックの影響を受けることになる。

9.7.3 ニューラルネットワークと GPU

同様に、ニューラルネットワークモデルでもデバイスを指定できる。次のコードは、モデルのパラメータを GPU 上に置く。

```
net = nn.Sequential(nn.LazyLinear(1))
net = net.to(device=try_gpu())
```

```
net = nn.Sequential()
net.add(nn.Dense(1))
net.initialize(ctx=try_gpu())
```

```
net = nn.Sequential([nn.Dense(1)])

key1, key2 = jax.random.split(jax.random.PRNGKey(0))
x = jax.random.normal(key1, (10,))  # Dummy input
params = net.init(key2, x)  # Initialization call
```

```
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    net = tf.keras.models.Sequential([
        tf.keras.layers.Dense(1)])
```

```
INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/
↪task:0/device:GPU:0', '/job:localhost/replica:0/task:0/device:GPU:1')
```

今後の章では、モデルを GPU 上で実行する方法の例をさらに多く見ていく。モデルがやや計算集約的になっていくからである。

たとえば、入力が GPU 上のテンソルであれば、モデルは同じ GPU 上で結果を計算する。

```
net(X)
```

```
tensor([[0.1320],
        [0.1320]], device='cuda:0', grad_fn=<AddmmBackward0>)
```

```
net(X)
```

```
array([[0.04995865],
       [0.04995865]], ctx=gpu(0))
```

```
net.apply(params, x)
```

```
Array([-1.2849933], dtype=float32)
```

```
net(X)
```

```
<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
array([[0.4419096],
       [0.4419096]], dtype=float32)>
```

モデルのパラメータが同じ GPU 上に保存されていることを確認してみよう。

```
net[0].weight.data.device
```

```
device(type='cuda', index=0)
```

```
net[0].weight.data().ctx
```

```
gpu(0)
```

```
print(jax.tree_util.tree_map(lambda x: x.device(), params))
```

```
FrozenDict({
    params: {
        layers_0: {
            bias: gpu(id=0),
            kernel: gpu(id=0),
        },
    },
})
```

```
net.layers[0].weights[0].device, net.layers[0].weights[1].device
```

```
('/job:localhost/replica:0/task:0/device:GPU:0',
 '/job:localhost/replica:0/task:0/device:GPU:0')
```

トレーナーが GPU をサポートするようにする。

```
@d2l.add_to_class(d2l.Trainer)  #@save
def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
```

(continues on next page)

(continued from previous page)

```
    self.save_hyperparameters()
    self.gpus = [d2l.gpu(i) for i in range(min(num_gpus, d2l.num_gpus()))]

@d2l.add_to_class(d2l.Trainer)  #@save
def prepare_batch(self, batch):
    if self.gpus:
        batch = [d2l.to(a, self.gpus[0]) for a in batch]
    return batch

@d2l.add_to_class(d2l.Trainer)  #@save
def prepare_model(self, model):
    model.trainer = self
    model.board.xlim = [0, self.max_epochs]
    if self.gpus:
        if tab.selected('mxnet'):
            model.collect_params().reset_ctx(self.gpus[0])
            model.set_scratch_params_device(self.gpus[0])
        if tab.selected('pytorch'):
            model.to(self.gpus[0])
    self.model = model
```

```
@d2l.add_to_class(d2l.Module)  #@save
def set_scratch_params_device(self, device):
    for attr in dir(self):
        a = getattr(self, attr)
        if isinstance(a, np.ndarray):
            with autograd.record():
                setattr(self, attr, a.as_in_ctx(device))
            getattr(self, attr).attach_grad()
        if isinstance(a, d2l.Module):
            a.set_scratch_params_device(device)
        if isinstance(a, list):
            for elem in a:
                elem.set_scratch_params_device(device)
```

```
@d2l.add_to_class(d2l.Trainer)  #@save
def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
    self.save_hyperparameters()
    self.gpus = [d2l.gpu(i) for i in range(min(num_gpus, d2l.num_gpus()))]

@d2l.add_to_class(d2l.Trainer)  #@save
def prepare_batch(self, batch):
    if self.gpus:
        batch = [d2l.to(a, self.gpus[0]) for a in batch]
    return batch
```

要するに、すべてのデータとパラメータが同じデバイス上にある限り、モデルを効率よく学習できる。次の章では、そのような例をいくつか見ていく。

9.7.4 まとめ

CPU や GPU など、保存や計算のためのデバイスを指定できる。デフォルトでは、データは主記憶に作成され、その後 CPU で計算に使われる。深層学習フレームワークでは、計算に必要なすべての入力データが CPU または同じ GPU 上にあることが求められる。データを不用意に移動すると、大きな性能低下を招くことがある。よくある間違いは次のようなものである。GPU 上で各ミニバッチの損失を計算し、それをコマンドラインでユーザーに報告したり（あるいは NumPy `ndarray` に記録したり）すると、グローバルインタプリタロックが発生してすべての GPU が停止する。ログ記録用のメモリを GPU 内に確保し、大きなログだけを移動するほうがはるかに良い。

9.7.5 演習

1. 大きな行列の積のような、より大きな計算タスクを試し、CPU と GPU の速度差を確認せよ。計算量が少ないタスクではどうだろうか。
2. モデルパラメータを GPU 上でどのように読み書きすべきだろうか。
3. 100×100 の行列どうしの行列積を 1000 回計算し、出力行列のフロベニウスノルムを 1 結果ずつ記録するのにかかる時間を測れ。GPU 上にログを保持し、最後の結果だけを転送する場合と比較せよ。
4. 2 枚の GPU 上で同時に 2 つの行列積を実行するのにかかる時間を測れ。1 枚の GPU 上で順番に計算する場合と比較せよ。ヒント: ほぼ線形スケーリングが見られるはずである。

10 畳み込みニューラルネットワーク

画像データは、モノクロであれカラーであれ、ピクセルの二次元グリッドとして表現される。したがって、各ピクセルはそれぞれ一つまたは複数の数値に対応する。これまで私たちは、この豊かな構造を無視し、ピクセル間の空間的な関係にかかわらず、画像を 平坦化して数値ベクトルとして扱ってきた。この、深く不満の残る方法は、得られた一次元ベクトルを全結合 MLP に入力するために必要であった。

これらのネットワークは特徴の順序に対して不変であるため、ピクセルの空間構造に対応する順序を保つ場合でも、MLP のパラメータを学習させる前に設計行列の列を並べ替える場合でも、同様の結果が得られる。理想的には、近接するピクセル同士が通常は関連しているという事前知識を活用して、画像データから学習するための効率的なモデルを構築したいところである。

この章では、まさにこの目的のために設計された強力なニューラルネットワーク群である 畳み込みニューラルネットワーク（CNN）を紹介する (LeCun *et al.*, 1995)。CNN ベースのアーキテクチャは、現在ではコンピュータビジョン分野の至るところで使われている。たとえば、Imagnet コレクション (Deng *et al.*, 2009) では、畳み込みニューラルネットワーク、略して Convnet の使用だけが、顕著な性能向上をもたらした (Krizhevsky *et al.*, 2012)。

現代の CNN は、通称でそう呼ばれるように、生物学、群論、そして十分な量の実験的試行錯誤から着想を得て設計された。正確なモデルを少ないサンプルで達成できるというサンプル効率の高さに加えて、CNN は計算効率にも優れている。これは、全結合アーキテクチャよりも必要なパラメータ数が少ないこと、そして畳み込みが GPU コア全体にわたって並列化しやすいことの両方による (Chetlur *et al.*, 2014)。その結果、実務家は可能な限り CNN を適用することが多く、さらに近年では、音声 (Abdel-Hamid *et al.*, 2014)、テキスト (Kalchbrenner *et al.*, 2014)、時系列解析 (LeCun *et al.*, 1995) のような、従来は再帰ニューラルネットワークが用いられてきた一次元の系列構造を持つタスクにおいても、有力な競争相手として台頭してきた。CNN を巧みに拡張することで、グラフ構造データ (Kipf and Welling, 2016) や推薦システムにも適用されるようになっている。

まず、畳み込みニューラルネットワークの動機をより深く掘り下げる。続いて、すべての畳み込みネットワークの骨格を成す基本操作を順に見ていく。これには、畳み込み層そのもの、パディングとストライドを含む細かな詳細、隣接する空間領域にまたがる情報を集約するために用いられるプーリング層、各層での複数チャネルの使用、そして現代的なアーキテクチャの構造に関する丁寧な議論が含まれる。章の最後では、現代の深層学習が台頭するはるか以前に

最初に成功裏に実用化された畳み込みネットワークである LeNet の完全な動作例を示す。次の章では、現代の実践者が一般的に用いる技法の大半を体現した、人気があり比較的新しいいくつかの CNN アーキテクチャについて、完全な実装を見ていく。

10.1 全結合層から畳み込みへ

今日に至るまで、これまで議論してきたモデルは、表形式データを扱う場合には依然として適切な選択肢である。表形式とは、データが例に対応する行と特徴量に対応する列から構成されていることを意味する。表形式データでは、私たちは探したいパターンが特徴量間の相互作用を含むかもしれないと予想できるが、特徴量がどのように相互作用するかについての構造を 先験的に仮定しない。

ときには、より洗練されたアーキテクチャの構築を導くための知識が本当に不足していることがある。このような場合、MLP が私たちにできる最善の方法かもしれない。しかし、高次元の知覚データでは、このような構造を持たないネットワークは扱いにくくなりがちである。

たとえば、猫と犬を見分けるというこれまでの例に戻ろう。十分に丁寧なデータ収集を行い、1 メガピクセルの写真からなる注釈付きデータセットを集めたとする。これは、ネットワークへの各入力が 100 万次元を持つことを意味する。たとえ 1000 個の隠れ次元へと強引に削減したとしても、$10^6 \times 10^3 = 10^9$ 個のパラメータを持つ全結合層が必要になる。大量の GPU、分散最適化の才能、そして並外れた忍耐力がない限り、このネットワークのパラメータを学習することは実行不可能かもしれない。

注意深い読者は、1 メガピクセルの解像度は必要ないのではないかとこの議論に異を唱えるかもしれない。しかし、10 万画素で済ませられるとしても、サイズ 1000 の隠れ層は、画像の良い表現を学習するのに必要な隠れユニット数を大幅に過小評価している。したがって、実用的なシステムでは依然として数十億個のパラメータが必要になるだろう。さらに、これほど多くのパラメータを当てはめて分類器を学習するには、膨大なデータセットの収集が必要になるかもしれない。それにもかかわらず、今日では人間もコンピュータも猫と犬をかなりうまく見分けることができ、これらの直感に反しているように見える。それは、画像には人間にも機械学習モデルにも活用できる豊かな構造があるからである。畳み込みニューラルネットワーク（CNN）は、自然画像に存在する既知の構造の一部を活用するために機械学習が取り入れた創造的な方法の一つである。

10.1.1 不変性

画像中の物体を検出したいとしよう。物体を認識するために用いる方法は、画像内でその物体がどこにあるかという正確な位置を過度に気にすべきではな

い、というのはもっともらしく思える。理想的には、私たちのシステムはこの知識を活用すべきである。豚は普通は飛ばず、飛行機は普通は泳がない。それでも、画像の上部に豚が現れたとしても、私たちはそれを豚だと認識できるべきである。ここでは、子どものゲーム「ウォーリーをさがせ」からいくらか着想を得ることができる（このゲーム自体も、Fig. 10.1.1 に示すような多くの実世界の模倣を生み出した）。このゲームは、さまざまな活動であふれた混沌とした場面がいくつも並んでいる。ウォーリーはそれぞれの場面のどこかに現れ、たいていはありそうもない場所にひそんでいる。読者の目標は彼を見つけることである。彼の特徴的な服装にもかかわらず、多くの気を散らす要素のために、これは驚くほど難しいことがある。しかし、ウォーリーがどのように見えるかは、ウォーリーがどこにいるかには依存しない。私たちは画像全体をウォーリー検出器で走査し、各パッチにスコアを割り当てて、そのパッチにウォーリーが含まれている確率を示すことができる。実際、多くの物体検出やセグメンテーションのアルゴリズムはこのアプローチに基づいている (Long *et al.*, 2015)。CNN は、この空間的不変性という考えを体系化し、より少ないパラメータで有用な表現を学習するためにこれを活用する。



Fig. 10.1.1 ウォーリーを見つけられるだろうか（画像提供: William Murphy (Infomatique)）?

ここで、これらの直感をより具体的にし、コンピュータビジョンに適したニューラルネットワークアーキテクチャの設計を導くいくつかの要件を列挙してみよう。

1. 最初の層では、画像内のどこに現れても、同じパッチに対してネットワークが同様に反応すべきである。この原理は平行移動不変性（または平行移動等価性）と呼ばれる。
2. ネットワークの最初の層は、画像の遠く離れた領域の内容に関係なく、局所領域に注目すべきである。これが局所性の原理である。最終的には、これらの局所表現を集約して画像全体レベルで予測を行える。
3. 進むにつれて、より深い層は、自然界における高次の視覚に似た形で、画像のより長距離の特徴を捉えられるべきである。

これが数学にどう対応するかを見ていこう。

10.1.2 MLP への制約

まず、2 次元画像 $\mathbf{X}$ を入力とし、その直後の隠れ表現 $\mathbf{H}$ も同様に行列として表される（コード上では 2 次元テンソルです）MLP を考える。ここで、$\mathbf{X}$ と $\mathbf{H}$ は同じ形状を持つとする。少し考えてみてほしい。ここでは、入力だけでなく隠れ表現も空間構造を持つと考える。

$\mathbf{X}$ の位置 (i, j) にある画素と $\mathbf{H}$ の位置 (i, j) にある画素をそれぞれ $[\mathbf{X}]_{i,j}$ と $[\mathbf{H}]_{i,j}$ で表す。したがって、各隠れユニットが各入力画素から入力を受け取るようにするには、（これまで MLP で行っていたように）重み行列を使う代わりに、パラメータを 4 階の重みテンソル W として表現することになる。$\mathbf{U}$ がバイアスを含むとすると、全結合層は形式的に次のように表せる。

$$\begin{aligned}[\mathbf{H}]_{i,j} &= [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l} \\ &= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}.\end{aligned} \tag{10.1.1}$$

W から V への切り替えは、今のところは単なる表記上の変更にすぎない。というのも、両方の 4 階テンソルの係数の間には 1 対 1 対応があるからである。添字 (k, l) を $k = i + a$ および $l = j + b$ となるように単に付け替えただけである。言い換えると、$[\mathsf{V}]_{i,j,a,b} = [\mathsf{W}]_{i,j,i+a,j+b}$ と置いている。添字 a と b は正負両方のオフセットを取り、画像全体を覆う。隠れ表現 $[\mathbf{H}]_{i,j}$ の任意の位置 (i, j) に対して、その値は、(i, j) を中心とする x の画素を $[\mathsf{V}]_{i,j,a,b}$ で重み付けして和を取ることで計算される。先へ進む前に、このパラメータ化で *1* 層に必要なパラメータ総数を考えてみよう。1000×1000 の画像（1 メガピクセル）が 1000×1000 の隠れ表現へ写像されるとする。この場合、10^{12} 個のパラメータが必要であり、これは現在のコンピュータでは到底扱えない。

平行移動不変性

では、上で述べた第一の原理、平行移動不変性 (Zhang *et al.*, 1988) を適用しよう。これは、入力 $\mathbf{X}$ のシフトが隠れ表現 $\mathbf{H}$ のシフトを単に引き起こすべきであることを意味する。これが可能なのは、V と $\mathbf{U}$ が実際には (i, j) に依存しない場合だけである。したがって、$[\mathsf{V}]_{i,j,a,b} = [\mathbf{V}]_{a,b}$ かつ $\mathbf{U}$ は定数、たとえば u となる。その結果、$\mathbf{H}$ の定義は次のように簡単になる。

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{10.1.2}$$

これは畳み込みである！私たちは実質的に、位置 (i, j) の近傍にある $(i + a, j + b)$ の画素を係数 $[\mathbf{V}]_{a,b}$ で重み付けして、$[\mathbf{H}]_{i,j}$ の値を得ている。$[\mathbf{V}]_{a,b}$ は画像内の位置に依存しなくなったため、$[\mathsf{V}]_{i,j,a,b}$ よりもはるかに少ない係数で済むことに注意してほしい。したがって、必要なパラメータ数はもはや 10^{12} ではなく、はるかに現実的な 4×10^6 になる。それでもなお、$a, b \in (-1000, 1000)$ への依存は残っている。要するに、私たちは大きく前進したのである。時間遅延ニ

ューラルネットワーク（TDNN）は、この考えを活用した最初期の例の一つである (Waibel *et al.*, 1989)。

局所性

次に、第二の原理である局所性を適用しよう。上で動機づけたように、位置 (i, j) の周囲からあまり遠くまで見なくても、$[\mathbf{H}]_{i,j}$ で何が起きているかを評価するのに必要な情報を得られるはずだと考える。これは、ある範囲 $|a| > \Delta$ または $|b| > \Delta$ の外側では、$[\mathbf{V}]_{a,b} = 0$ とすべきことを意味する。同値に、$[\mathbf{H}]_{i,j}$ は次のように書き直せる。

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{10.1.3}$$

これにより、パラメータ数は 4×10^6 から $4\Delta^2$ に減る。ここで、Δ は通常 10 より小さい。このようにして、私たちはさらに 4 桁のオーダーでパラメータ数を削減した。(10.1.3) は、要するに畳み込み層と呼ばれるものである。畳み込みニューラルネットワーク（CNN）は、畳み込み層を含むニューラルネットワークの特別な一族である。深層学習研究コミュニティでは、$\mathbf{V}$ は畳み込みカーネル、フィルタ、あるいは単に学習可能パラメータとしての層の重みと呼ばれる。

以前は、画像処理ネットワークのたった 1 層を表現するのに数十億個のパラメータが必要だったかもしれないが、今では通常、入力や隠れ表現の次元を変えずに、数百個程度で済む。この劇的なパラメータ削減の代償として、特徴が平行移動不変になり、各隠れ活性の値を決める際に層が局所情報しか取り込めなくなることがある。すべての学習は、帰納バイアスを課すことに依存している。そのバイアスが現実と一致していれば、少ないサンプルで学習でき、未見データにもよく一般化するモデルが得られる。しかしもちろん、そのバイアスが現実と一致しなければ、たとえば画像が実際には平行移動不変でなかったなら、モデルは訓練データにさえうまく適合できないかもしれない。

この劇的なパラメータ削減は、最後の要件へと私たちを導く。すなわち、より深い層は画像のより大きく複雑な側面を表現すべきだということである。これは、非線形性と畳み込み層を何度も交互に重ねることで実現できる。

10.1.3 畳み込み

なぜ (10.1.3) が畳み込みと呼ばれるのかを、簡単に復習しよう。数学では、2 つの関数 (Rudin, 1973)、たとえば $f, g : \mathbb{R}^d \to \mathbb{R}$ の間の畳み込みは次のように定義される。

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z}) g(\mathbf{x} - \mathbf{z}) d\mathbf{z}. \tag{10.1.4}$$

つまり、一方の関数を「反転」して $\mathbf{x}$ だけ平行移動したときの、f と g の重なりを測っている。離散的な対象を扱う場合には、積分は和に変わる。たとえば、

添字が $\mathbb{Z}$ を走る二乗和可能な無限次元ベクトルの集合からのベクトルについては、次の定義が得られる。

$$(f * g)(i) = \sum_a f(a)g(i-a). \tag{10.1.5}$$

2 次元テンソルでは、f に対して添字 (a, b)、g に対して添字 $(i-a, j-b)$ を用いる対応する和を持つ。

$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i-a, j-b). \tag{10.1.6}$$

これは (10.1.3) に似ているが、1 つ大きな違いがある。$(i+a, j+b)$ を使う代わりに、差を使っているのである。ただし、この違いは主として表記上のものにすぎない。というのも、(10.1.3) と (10.1.6) の間でいつでも記法を対応づけられるからである。元の (10.1.3) の定義は、より正確には 相互相関を表している。これについては次の節で戻ってくる。

10.1.4 チャネル

ウォーリー検出器に戻って、これがどう見えるかを見てみよう。畳み込み層は、Fig. 10.1.2 に示すように、所定のサイズの窓を取り出し、フィルタ V に従って強度に重みを付ける。私たちは、どこで「ウォーリーらしさ」が最も高いかに応じて、隠れ層表現にピークが現れるようなモデルを学習したいと考えるかもしれない。



Fig. 10.1.2 ウォーリーを検出する（画像提供: William Murphy (Infomatique)）。

この方法には 1 つ問題がある。これまで私たちは、画像が 3 つのチャネル、すなわち赤・緑・青から成ることをうっかり無視していた。要するに、画像は 2 次元の対象ではなく、高さ・幅・チャネルで特徴づけられる 3 階テンソルであり、たとえば $1024 \times 1024 \times 3$ ピクセルの形状を持つ。最初の 2 つの軸は空間的関係に関わるが、3 つ目の軸は各画素位置に多次元表現を割り当てるものと見なせる。したがって、X は $[\mathsf{X}]_{i,j,k}$ としてインデックス付けする。畳み込みフィル

タもそれに応じて適応しなければならない。$[\mathbf{V}]_{a,b}$ の代わりに、今度は $[\mathsf{V}]_{a,b,c}$ を持つ。

さらに、入力が 3 階テンソルであるのと同様に、隠れ表現も同様に 3 階テンソル H として定式化するのが良い考えであることがわかる。言い換えると、各空間位置に対応する単一の隠れ表現を持つのではなく、各空間位置に対応する隠れ表現のベクトル全体を持ちたいのである。隠れ表現は、いくつもの 2 次元グリッドを積み重ねたものと考えられる。入力と同様に、これらはしばしばチャネルと呼ばれる。また、各層が次の層のために学習された特徴の空間化された集合を提供することから、特徴マップとも呼ばれる。直感的には、入力に近い低い層では、あるチャネルはエッジの認識に特化し、別のチャネルはテクスチャの認識に特化する、と考えられる。

入力（X）と隠れ表現（H）の両方で複数チャネルを扱うために、V に 4 つ目の座標を追加できる: $[\mathsf{V}]_{a,b,c,d}$。すべてをまとめると、次のようになる。

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} [\mathsf{V}]_{a,b,c,d} [\mathsf{X}]_{i+a,j+b,c}, \tag{10.1.7}$$

ここで d は隠れ表現 H における出力チャネルを表す。続く畳み込み層は、3 階テンソル H を入力として受け取る。一般性のために、(10.1.7) を複数チャネルに対する畳み込み層の定義とみなし、V をその層のカーネルまたはフィルタとする。

まだ対処すべき操作はたくさんある。たとえば、すべての隠れ表現を 1 つの出力にどうまとめるか、たとえば画像のどこかにウォーリーがいるのかを判定する方法を考える必要がある。また、どのように効率よく計算するか、複数層をどう組み合わせるか、適切な活性化関数は何か、そして実際に有効なネットワークを得るためにどのような設計選択が妥当かも決める必要がある。これらの問題については、本章の残りで扱う。

10.1.5 要約と考察

この節では、畳み込みニューラルネットワークの構造を第一原理から導出した。これが CNN の発明に至った経路だったかは不明だが、少なくとも低次のレベルでは、画像処理やコンピュータビジョンのアルゴリズムがどのように動作すべきかについて妥当な原理を適用すると、CNN が正しい選択であることがわかり、満足感がある。特に、画像における平行移動不変性は、画像のすべてのパッチが同じように扱われることを意味する。局所性とは、対応する隠れ表現を計算するために画素の小さな近傍だけを使うことを意味する。CNN に関する最も初期の文献の一部は、ネオコグニトロンの形で現れている (Fukushima, 1982)。

推論の中で出てきた第二の原理は、ある仮定がモデルに成り立つ限り、表現力を制限せずに関数クラスのパラメータ数をどう減らすかということであった。この制約の結果として複雑さが劇的に減少し、計算的にも統計的にも実行不可能だった問題が扱えるモデルへと変わった。

チャネルを追加することで、局所性と平行移動不変性によって畳み込みカーネルに課された制約のために失われた複雑さの一部を取り戻すことができた。赤・緑・青以外のチャネルを追加するのはごく自然なことである。特に農業や気象学で用いられる多くの衛星画像は、数十から数百のチャネルを持ち、代わりにハイパースペクトル画像を生成する。それらは多くの異なる波長に関するデータを報告する。以下では、畳み込みを効果的に使ってそれが作用する画像の次元を操作する方法、位置ベースの表現からチャネルベースの表現へ移る方法、そして多数のカテゴリを効率よく扱う方法を見ていく。

10.1.6 演習

1. 畳み込みカーネルのサイズが $\Delta = 0$ であると仮定する。このとき、畳み込みカーネルが各チャネル集合ごとに独立に MLP を実装することを示せ。これは Network in Network アーキテクチャにつながる (Lin *et al.*, 2013)。
2. 音声データはしばしば 1 次元の系列として表される。
    1. 音声に対して局所性と平行移動不変性を課したいのは、どのような場合だろうか。
    2. 音声に対する畳み込み演算を導出せよ。
    3. コンピュータビジョンと同じ道具で音声を扱えるだろうか。ヒント: スペクトログラムを使え。
3. 平行移動不変性は、結局のところ良い考えではないかもしれない。なぜだろうか。例を挙げよ。
4. 畳み込み層はテキストデータにも適用できると思うか？言語ではどのような問題に直面するだろうか。
5. 物体が画像の境界にあるとき、畳み込みでは何が起こるだろうか。
6. 畳み込みが対称、すなわち $f * g = g * f$ であることを証明せよ。

10.2 画像のための畳み込み

ここまでで、畳み込み層が理論上どのように動作するかを理解した。いよいよ、実際にどのように機能するかを見ていこう。画像データの構造を効率よく探索するためのアーキテクチャとして畳み込みニューラルネットワークを考えるという動機に基づき、ここでは引き続き画像を例として用いる。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import autograd, np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

10.2.1 相互相関演算

厳密に言えば、畳み込み層という呼び方はやや不正確であり、そこで表現される演算は、より正確には相互相関と呼ぶべきものである。Section 10.1 における畳み込み層の説明に基づくと、このような層では、入力テンソルとカーネルテンソルが相互相関演算を通じて結合され、出力テンソルが生成される。

ここではまずチャネルを無視して、2 次元データと隠れ表現でこれがどのように機能するかを見てみよう。Fig. 10.2.1 では、入力は高さ 3、幅 3 の 2 次元テンソルである。テンソルの形状は 3×3 または $(3, 3)$ と表す。カーネルの高さと幅はいずれも 2 である。カーネルウィンドウ（または 畳み込みウィンドウ）の形状は、カーネルの高さと幅によって与えられる（ここでは 2×2 である）。

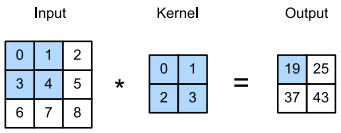


Fig. 10.2.1 2 次元相互相関演算。網掛け部分は最初の出力要素であり、出力計算に使われる入力テンソルとカーネルテンソルの要素も示している:
$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

2 次元相互相関演算では、まず畳み込みウィンドウを入力テンソルの左上隅に置き、そこから入力テンソル上を左から右へ、上から下へとスライドさせる。畳み込みウィンドウがある位置に移動すると、そのウィンドウに含まれる入力部分テンソルとカーネルテンソルを要素ごとに掛け合わせ、その結果のテンソルを総和して 1 つのスカラー値を得る。この結果が、対応する位置における出力テンソルの値になる。ここでは、出力テンソルの高さは 2、幅は 2 であり、4 つ

の要素は 2 次元相互相関演算から次のように得られる。

$$\begin{aligned}0\times 0+1\times 1+3\times 2+4\times 3=19,\\1\times 0+2\times 1+4\times 2+5\times 3=25,\\3\times 0+4\times 1+6\times 2+7\times 3=37,\\4\times 0+5\times 1+7\times 2+8\times 3=43.\end{aligned} \tag{10.2.1}$$

各軸に沿って見ると、出力サイズは入力サイズよりわずかに小さくなっている。カーネルの幅と高さが 1 より大きいため、カーネルが画像内に完全に収まる位置でしか相互相関を正しく計算できない。そのため、出力サイズは入力サイズ $n_\mathrm{h}\times n_\mathrm{w}$ から畳み込みカーネルのサイズ $k_\mathrm{h}\times k_\mathrm{w}$ を引いて、

$$(n_\mathrm{h}-k_\mathrm{h}+1)\times(n_\mathrm{w}-k_\mathrm{w}+1). \tag{10.2.2}$$

となる。

これは、畳み込みカーネルを画像上で「ずらす」ための十分な空間が必要だからである。後で、画像の境界の周囲にゼロを埋め込むことで、カーネルをずらすための十分な空間を確保し、サイズを変えずに保つ方法を見る。次に、この処理を `corr2d` 関数として実装する。この関数は入力テンソル `X` とカーネルテンソル `K` を受け取り、出力テンソル `Y` を返す。

```
def corr2d(X, K):  #@save
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = d2l.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = d2l.reduce_sum((X[i: i + h, j: j + w] * K))
    return Y
```

```
def corr2d(X, K):  #@save
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = d2l.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = d2l.reduce_sum((X[i: i + h, j: j + w] * K))
    return Y
```

```
def corr2d(X, K):  #@save
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = jnp.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y = Y.at[i, j].set((X[i:i + h, j:j + w] * K).sum())
    return Y
```

```
def corr2d(X, K):  #@save
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = tf.Variable(tf.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1)))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j].assign(tf.reduce_sum(
                X[i: i + h, j: j + w] * K))
    return Y
```

入力テンソル X とカーネルテンソル K を Fig. 10.2.1 から構成し、上の実装が 2 次元相互相関演算の出力を正しく与えることを確認できる。

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

```
tensor([[19., 25.],
        [37., 43.]])
```

10.2.2 畳み込み層

畳み込み層は、入力とカーネルの相互相関を計算し、さらにスカラーのバイアスを加えて出力を生成する。畳み込み層の 2 つのパラメータは、カーネルとスカラーのバイアスである。畳み込み層に基づくモデルを学習するときには、全結合層の場合と同様に、通常カーネルをランダムに初期化する。

ここで、上で定義した corr2d 関数に基づいて、2 次元畳み込み層を実装する準備が整った。__init__ コンストラクタメソッドでは、weight と bias を 2 つのモデルパラメータとして宣言する。順伝播メソッドは corr2d 関数を呼び出し、バイアスを加える。

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

```
class Conv2D(nn.Block):
    def __init__(self, kernel_size, **kwargs):
        super().__init__(**kwargs)
        self.weight = self.params.get('weight', shape=kernel_size)
        self.bias = self.params.get('bias', shape=(1,))

    def forward(self, x):
        return corr2d(x, self.weight.data()) + self.bias.data()
```

```
class Conv2D(nn.Module):
    kernel_size: int

    def setup(self):
        self.weight = nn.param('w', nn.initializers.uniform, self.kernel_size)
        self.bias = nn.param('b', nn.initializers.zeros, 1)

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

```
class Conv2D(tf.keras.layers.Layer):
    def __init__(self):
        super().__init__()

    def build(self, kernel_size):
        initializer = tf.random_normal_initializer()
        self.weight = self.add_weight(name='w', shape=kernel_size,
                                      initializer=initializer)
        self.bias = self.add_weight(name='b', shape=(1, ),
                                    initializer=initializer)

    def call(self, inputs):
        return corr2d(inputs, self.weight) + self.bias
```

$h \times w$ 畳み込み、または $h \times w$ 畳み込みカーネルでは、畳み込みカーネルの高さと幅はそれぞれ h と w である。また、$h \times w$ 畳み込みカーネルをもつ畳み込み層を、単に $h \times w$ 畳み込み層とも呼ぶ。

10.2.3 画像における物体のエッジ検出

画素の変化位置を見つけることで、畳み込み層の簡単な応用である、画像中の物体のエッジ検出を少し見てみよう。まず、6×8 ピクセルの「画像」を構成する。中央の 4 列は黒（0）で、それ以外は白（1）である。

```
X = d2l.ones((6, 8))
X[:, 2:6] = 0
X
```

```
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
```

```
X = d2l.ones((6, 8))
X[:, 2:6] = 0
X
```

```
array([[1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.]])
```

```
X = jnp.ones((6, 8))
X = X.at[:, 2:6].set(0)
X
```

```
Array([[1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.]], dtype=float32)
```

```
X = tf.Variable(tf.ones((6, 8)))
X[:, 2:6].assign(tf.zeros(X[:, 2:6].shape))
X
```

```
<tf.Variable 'Variable:0' shape=(6, 8) dtype=float32, numpy=
array([[1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.]], dtype=float32)>
```

次に、高さ 1、幅 2 のカーネル `K` を構成する。入力に対して相互相関演算を行うと、水平方向に隣接する要素が同じなら出力は 0 になる。そうでなければ、出力は 0 ではない。このカーネルは有限差分演算子の特殊な場合であることに注意してほしい。位置 (i, j) では $x_{i,j} - x_{(i+1),j}$ を計算する。つまり、水平方向に隣接する画素の値の差を計算している。これは、水平方向の 1 階微分の離散近似である。実際、関数 $f(i, j)$ に対して、その微分は $-\partial_i f(i, j) = \lim_{\epsilon \to 0} \frac{f(i,j) - f(i+\epsilon, j)}{\epsilon}$ である。これが実際にどのように働くか見てみよう。

```
K = d2l.tensor([[1.0, -1.0]])
```

入力 `X` とカーネル `K` を用いて相互相関演算を行う準備ができた。ご覧のとおり、白から黒へのエッジでは 1 を検出し、黒から白へのエッジでは -1 を検出する。それ以外の出力はすべて 0 である。

```
Y = corr2d(X, K)
Y
```

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

次に、このカーネルを転置した画像に適用できる。予想どおり、結果は 0 になる。カーネル `K` は垂直エッジのみを検出する。

```
corr2d(d2l.transpose(X), K)
```

```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
```

10.2.4 カーネルの学習

有限差分 `[1, -1]` によるエッジ検出器を設計するのは、まさにそれが必要なものだと分かっているなら簡潔で美しい方法である。しかし、より大きなカーネルを考えたり、畳み込み層を何層も重ねたりすると、各フィルタが何をすべきかを手作業で正確に指定するのは不可能かもしれない。

では、入力と出力のペアだけを見て、`X` から `Y` を生成したカーネルを学習できるかを見てみよう。まず畳み込み層を構成し、そのカーネルをランダムなテンソルとして初期化する。次に、各反復で二乗誤差を用いて `Y` と畳み込み層の出力を比較する。その後、勾配を計算してカーネルを更新できる。簡単のため、以下では 2 次元畳み込み層の組み込みクラスを使い、バイアスは無視する。

```
# Construct a two-dimensional convolutional layer with 1 output channel and a
# kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
conv2d = nn.LazyConv2d(1, kernel_size=(1, 2), bias=False)

# The two-dimensional convolutional layer uses four-dimensional input and
# output in the format of (example, channel, height, width), where the batch
# size (number of examples in the batch) and the number of channels are both 1
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2  # Learning rate

for i in range(10):
    Y_hat = conv2d(X)
    l = (Y_hat - Y) ** 2
    conv2d.zero_grad()
```

(continues on next page)

(continued from previous page)

```
    l.sum().backward()
    # Update the kernel
    conv2d.weight.data[:] -= lr * conv2d.weight.grad
    if (i + 1) % 2 == 0:
        print(f'epoch {i + 1}, loss {l.sum():.3f}')
```

```
epoch 2, loss 14.442
epoch 4, loss 3.970
epoch 6, loss 1.299
epoch 8, loss 0.477
epoch 10, loss 0.186
```

```
# Construct a two-dimensional convolutional layer with 1 output channel and a
# kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
conv2d = nn.Conv2D(1, kernel_size=(1, 2), use_bias=False)
conv2d.initialize()

# The two-dimensional convolutional layer uses four-dimensional input and
# output in the format of (example, channel, height, width), where the batch
# size (number of examples in the batch) and the number of channels are both 1
X = X.reshape(1, 1, 6, 8)
Y = Y.reshape(1, 1, 6, 7)
lr = 3e-2  # Learning rate

for i in range(10):
    with autograd.record():
        Y_hat = conv2d(X)
        l = (Y_hat - Y) ** 2
    l.backward()
    # Update the kernel
    conv2d.weight.data()[:] -= lr * conv2d.weight.grad()
    if (i + 1) % 2 == 0:
        print(f'epoch {i + 1}, loss {float(l.sum()):.3f}')
```

```
epoch 2, loss 4.949
epoch 4, loss 0.831
epoch 6, loss 0.140
epoch 8, loss 0.024
epoch 10, loss 0.004
[07:06:32] ../src/base.cc:48: GPU context requested, but no GPUs found.
```

```
# Construct a two-dimensional convolutional layer with 1 output channel and a
# kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
conv2d = nn.Conv(1, kernel_size=(1, 2), use_bias=False, padding='VALID')

# The two-dimensional convolutional layer uses four-dimensional input and
# output in the format of (example, height, width, channel), where the batch
# size (number of examples in the batch) and the number of channels are both 1
X = X.reshape((1, 6, 8, 1))
Y = Y.reshape((1, 6, 7, 1))
```

(continues on next page)

(continued from previous page)

```
lr = 3e-2  # Learning rate

params = conv2d.init(jax.random.PRNGKey(d2l.get_seed()), X)

def loss(params, X, Y):
    Y_hat = conv2d.apply(params, X)
    return ((Y_hat - Y) ** 2).sum()

for i in range(10):
    l, grads = jax.value_and_grad(loss)(params, X, Y)
    # Update the kernel
    params = jax.tree_map(lambda p, g: p - lr * g, params, grads)
    if (i + 1) % 2 == 0:
        print(f'epoch {i + 1}, loss {l:.3f}')
```

```
epoch 2, loss 8.136
epoch 4, loss 1.726
epoch 6, loss 0.437
epoch 8, loss 0.134
epoch 10, loss 0.047
```

```
# Construct a two-dimensional convolutional layer with 1 output channel and a
# kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
conv2d = tf.keras.layers.Conv2D(1, (1, 2), use_bias=False)

# The two-dimensional convolutional layer uses four-dimensional input and
# output in the format of (example, height, width, channel), where the batch
# size (number of examples in the batch) and the number of channels are both 1
X = tf.reshape(X, (1, 6, 8, 1))
Y = tf.reshape(Y, (1, 6, 7, 1))
lr = 3e-2  # Learning rate

Y_hat = conv2d(X)
for i in range(10):
    with tf.GradientTape(watch_accessed_variables=False) as g:
        g.watch(conv2d.weights[0])
        Y_hat = conv2d(X)
        l = (abs(Y_hat - Y)) ** 2
        # Update the kernel
        update = tf.multiply(lr, g.gradient(l, conv2d.weights[0]))
        weights = conv2d.get_weights()
        weights[0] = conv2d.weights[0] - update
        conv2d.set_weights(weights)
        if (i + 1) % 2 == 0:
            print(f'epoch {i + 1}, loss {tf.reduce_sum(l):.3f}')
```

```
epoch 2, loss 7.524
epoch 4, loss 1.912
epoch 6, loss 0.587
epoch 8, loss 0.207
epoch 10, loss 0.079
```

10 回の反復の後で誤差が小さな値まで下がっていることに注意してほしい。では、学習したカーネルテンソルを見てみよう。

```
d2l.reshape(conv2d.weight.data, (1, 2))
```

```
tensor([[ 0.9414, -1.0291]])
```

```
d2l.reshape(conv2d.weight.data(), (1, 2))
```

```
array([[ 0.9895   , -0.9873705]])
```

```
params['params']['kernel'].reshape((1, 2))
```

```
Array([[ 1.0077564, -0.965434 ]], dtype=float32)
```

```
d2l.reshape(conv2d.get_weights()[0], (1, 2))
```

```
<tf.Tensor: shape=(1, 2), dtype=float32, numpy=array([[ 0.9601698, -1.016949
↪]], dtype=float32)>
```

実際、学習されたカーネルテンソルは、先ほど定義したカーネルテンソル K に驚くほど近いものになっている。

10.2.5 相互相関と畳み込み

Section 10.1 で述べた、相互相関演算と畳み込み演算の対応関係を思い出してほしい。ここでは引き続き 2 次元畳み込み層を考える。このような層が、相互相関ではなく (10.1.6) で定義された厳密な畳み込み演算を行うとしたらどうなるだろうか。厳密な 畳み込み演算の出力を得るには、2 次元カーネルテンソルを水平方向と垂直方向の両方で反転させたうえで、入力テンソルに対して 相互相関演算を行えばよいだけである。

注目すべきことに、深層学習ではカーネルはデータから学習されるため、畳み込み層が厳密な畳み込み演算を行うか、相互相関演算を行うかにかかわらず、その出力は変わらない。

これを説明するために、畳み込み層が 相互相関を行い、Fig. 10.2.1 におけるカーネルを学習したとしよう。ここではそれを行列 $\mathbf{K}$ と表す。他の条件が同じだと仮定すると、この層が厳密な 畳み込みを行う場合、学習されるカーネル $\mathbf{K}'$ は、$\mathbf{K}'$ を水平方向と垂直方向の両方で反転させたものが $\mathbf{K}$ と同じになる。言い換えると、畳み込み層が Fig. 10.2.1 の入力と $\mathbf{K}'$ に対して厳密な 畳み込みを行うと、Fig. 10.2.1 と同じ出力（入力と $\mathbf{K}$ の相互相関）が得られる。

深層学習の文献における標準的な用語法に従い、厳密には少し異なるにもかかわらず、ここでも相互相関演算を畳み込みと呼び続ける。さらに、要素という用語は、層の表現や畳み込みカーネルを表す任意のテンソルのエントリ（または成分）を指すために用いる。

10.2.6 特徴マップと受容野

Section 10.1.4 で述べたように、Fig. 10.2.1 における畳み込み層の出力は、空間次元（たとえば幅と高さ）における学習された表現（特徴）を後続層へ渡すものとみなせるため、特徴マップと呼ばれることがある。CNN では、ある層の任意の要素 x に対して、その 受容野とは、順伝播の間に x の計算に影響を与えうる（すべての前の層からの）要素全体を指す。受容野は入力の実際のサイズより大きくなることもある点に注意してほしい。

受容野を説明するために、引き続き Fig. 10.2.1 を使おう。2×2 の畳み込みカーネルが与えられたとき、網掛けされた出力要素（値 19）の受容野は、入力の網掛け部分にある 4 つの要素である。ここで 2×2 の出力を $\mathbf{Y}$ と表し、$\mathbf{Y}$ を入力として受け取り、単一要素 z を出力する、追加の 2×2 畳み込み層をもつより深い CNN を考える。この場合、$\mathbf{Y}$ 上での z の受容野には $\mathbf{Y}$ の 4 要素すべてが含まれ、入力上での受容野には 9 個の入力要素すべてが含まれる。したがって、特徴マップ内の任意の要素が、より広い領域にわたる入力特徴を検出するためにより大きな受容野を必要とするなら、より深いネットワークを構成すればよいのである。

受容野という名称は神経生理学に由来する。さまざまな刺激を用いた多様な動物に対する一連の実験 (Hubel and Wiesel, 1959, Hubel and Wiesel, 1962, Hubel and Wiesel, 1968) は、いわゆる視覚野の反応を調べた。概して、低い層はエッジや関連する形状に反応することが分かった。その後、Field (1987) は、自然画像に対してこの効果を、まさに畳み込みカーネルと呼ぶほかないもので示した。その顕著な類似性を示すために、重要な図を Fig. 10.2.2 に再掲する。

実際、この関係は、画像分類タスクで学習されたネットワークのより深い層で計算される特徴にも当てはまることが、たとえば Kuzovkin *et al.* (2018) によって示されている。要するに、畳み込みは生物学においてもコードにおいても、コンピュータビジョンにとって非常に強力な道具であることが証明されてきた。そのため、深層学習における最近の成功を（今となっては）畳み込みが予告していたとしても、驚くにはあたらない。

10.2.7 まとめ

畳み込み層に必要な中心的計算は相互相関演算である。単純な入れ子の for ループだけでその値を計算できることを見た。入力チャネルと出力チャネルが複数ある場合には、チャネル間で行列-行列演算を行う。見てきたように、計算は単純で、何よりも非常に 局所的である。これにより、ハードウェア最適化の余地が大きくなり、近年のコンピュータビジョンの多くの成果はそれによってのみ可能になっている。結局のところ、畳み込みの最適化では、メモリよりも高速計算にチップ設計者が投資できることを意味する。これは他の用途に対して最

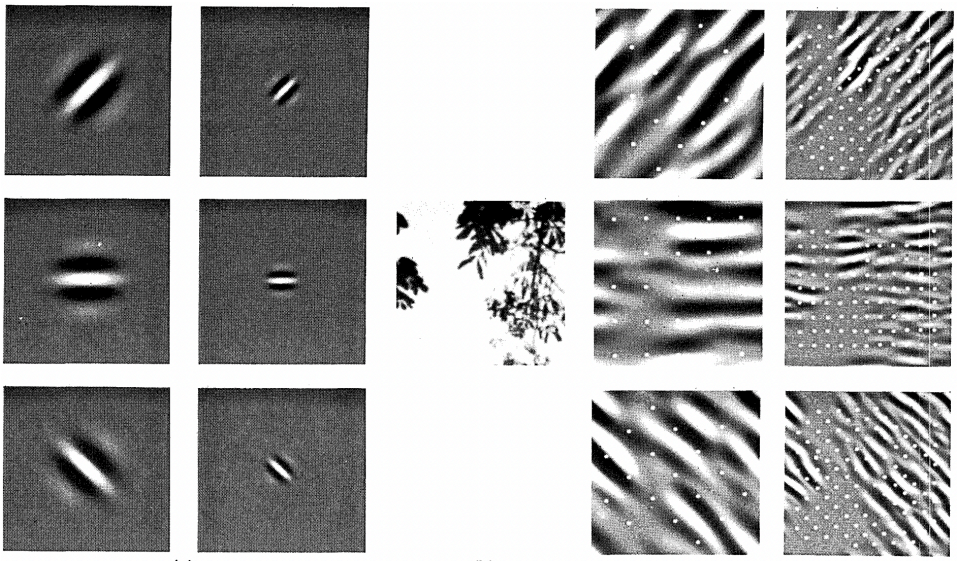


Fig. 10.2.2 Figure and caption taken from Field (1987): An example of coding with six different channels. (Left) Examples of the six types of sensor associated with each channel. (Right) Convolution of the image in (Middle) with the six sensors shown in (Left). The response of the individual sensors is determined by sampling these filtered images at a distance proportional to the size of the sensor (shown with dots). This diagram shows the response of only the even symmetric sensors.

適な設計をもたらすとは限らないが、普及しやすく手頃なコンピュータビジョンへの扉を開く。

畳み込みそのものについて言えば、エッジや線の検出、画像のぼかし、鮮鋭化など、多くの目的に使える。最も重要なのは、統計家（あるいはエンジニア）が適切なフィルタを発明する必要がないことである。その代わりに、データからそれらを単に 学習すればよいのである。これにより、特徴量設計の経験則は、証拠に基づく統計へと置き換えられる。最後に、そして実に喜ばしいことに、これらのフィルタは深いネットワークを構築するうえで有利であるだけでなく、脳における受容野や特徴マップにも対応している。これは、私たちが正しい方向に進んでいるという自信を与えてくれる。

10.2.8 演習

1. 対角線のエッジをもつ画像 X を構成せよ。
    1. この節のカーネル K を適用するとどうなるか。
    2. X を転置するとどうなるか。
    3. K を転置するとどうなるか。
2. いくつかのカーネルを手作業で設計せよ。

1. 方向ベクトル $\mathbf{v} = (v_1, v_2)$ が与えられたとき、$\mathbf{v}$ に直交するエッジ、すなわち $(v_2, -v_1)$ 方向のエッジを検出するエッジ検出カーネルを導出せよ。
2. 2 階微分の有限差分演算子を導出せよ。それに対応する畳み込みカーネルの最小サイズはいくつか。画像中のどの構造がそれに最も強く反応するか。
3. ぼかしカーネルをどのように設計するか。そのようなカーネルを使いたいのはなぜか。
4. d 階微分を得るためのカーネルの最小サイズはいくつか。

3. ここで作成した `Conv2D` クラスの勾配を自動的に求めようとすると、どのようなエラーメッセージが表示されるか。
4. 入力テンソルとカーネルテンソルを変形することで、相互相関演算を行列乗算としてどのように表現できるか。

10.3 パディングとストライド

Fig. 10.2.1 の畳み込みの例を思い出してほしい。入力の高さと幅はともに 3 で、畳み込みカーネルの高さと幅はともに 2 であった。その結果、出力表現の次元は 2×2 になった。入力の形状が $n_\mathrm{h} \times n_\mathrm{w}$ で、畳み込みカーネルの形状が $k_\mathrm{h} \times k_\mathrm{w}$ だとすると、出力の形状は $(n_\mathrm{h} - k_\mathrm{h} + 1) \times (n_\mathrm{w} - k_\mathrm{w} + 1)$ になる。つまり、畳み込みを適用できる画素が尽きるまでしか、畳み込みカーネルを移動できないのである。

以下では、パディングやストライド付き畳み込みを含むいくつかの手法を見ていく。これらは出力サイズをより柔軟に制御するためのものである。その動機として、カーネルの幅と高さは通常 1 より大きいため、多くの畳み込みを連続して適用すると、出力は入力よりかなり小さくなりがちであることに注意してほしい。240×240 ピクセルの画像から始めると、5×5 の畳み込みを 10 層重ねることで画像は 200×200 ピクセルまで縮小され、画像の 30% が切り落とされるうえに、元画像の境界にある興味深い情報もすべて失われてしまう。パディングは、この問題に対処するための最も一般的な手段である。別の状況では、たとえば元の入力解像度が扱いにくいと感じる場合などに、次元を大幅に削減したいこともある。そのような場合に役立つ一般的な手法が ストライド付き畳み込みである。

```
import torch
from torch import nn
```

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
import tensorflow as tf
```

10.3.1 パディング

上で述べたように、畳み込み層を適用するときの厄介な問題の 1 つは、画像の周辺部の画素が失われがちなことである。畳み込みカーネルのサイズと画像内の位置に応じた画素の利用状況を示す Fig. 10.3.1 を考えてみてほしい。四隅の画素はほとんど使われない。

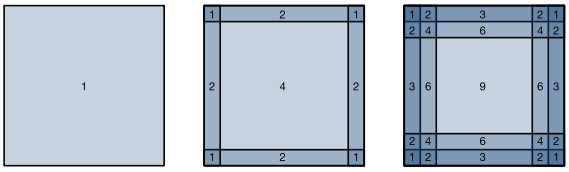


Fig. 10.3.1 それぞれサイズ 1×1、2×2、3×3 の畳み込みにおける画素の利用状況。

通常は小さなカーネルを使うので、1 回の畳み込みで失われる画素数は少なくても、畳み込み層を何層も重ねるとその損失は積み重なっていく。この問題に対する単純な解決策は、入力画像の境界の周囲に余分な埋め草の画素を追加して、画像の有効サイズを大きくすることである。通常、余分な画素の値は 0 に設定する。Fig. 10.3.2 では、3×3 の入力にパディングを施して、サイズを 5×5 に拡張している。それに対応する出力は 4×4 行列に増える。網掛け部分は最初の出力要素と、その出力計算に使われる入力テンソルおよびカーネルテンソルの要素を表している: $0\times0+0\times1+0\times2+0\times3=0$。

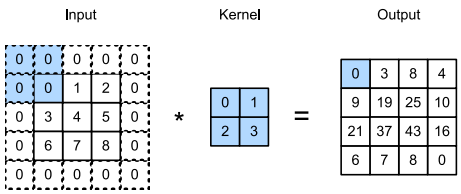


Fig. 10.3.2 パディング付きの 2 次元相互相関。

一般に、合計 p_h 行のパディング（おおよそ上側に半分、下側に半分）と、合計 p_w 列のパディング（おおよそ左側に半分、右側に半分）を追加すると、出力の

形状は

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1). \tag{10.3.1}$$

となる。

これは、出力の高さと幅がそれぞれ p_h と p_w だけ増えることを意味する。

多くの場合、入力と出力の高さと幅を同じにするために、$p_h = k_h - 1$ および $p_w = k_w - 1$ と設定したくなる。こうすると、ネットワークを構築するときに各層の出力形状を予測しやすくなる。ここで k_h が奇数だと仮定すると、高さ方向の両側に $p_h/2$ 行ずつパディングする。k_h が偶数の場合の 1 つの方法は、入力の上側に $\lceil p_h/2 \rceil$ 行、下側に $\lfloor p_h/2 \rfloor$ 行をパディングすることである。幅方向についても同様に両側をパディングする。

CNN では、1、3、5、7 などの奇数の高さと幅をもつ畳み込みカーネルが一般的に使われる。奇数サイズのカーネルを選ぶ利点は、上下に同じ行数、左右に同じ列数をパディングしながら、次元を保てることである。

さらに、このように奇数カーネルを使い、次元を正確に保つようにパディングする慣習には、事務的な利点もある。任意の 2 次元テンソル `X` について、カーネルのサイズが奇数で、すべての辺でのパディング行数と列数が同じであり、その結果として入力と同じ高さと幅をもつ出力が得られるとき、出力 `Y[i, j]` は、ウィンドウの中心が `X[i, j]` に来るようにして入力と畳み込みカーネルの相互相関を計算したものだと分かる。

次の例では、高さと幅が 3 の 2 次元畳み込み層を作成し、すべての辺に 1 ピクセルのパディングを適用する。高さと幅が 8 の入力を与えると、出力の高さと幅も 8 になることが分かる。

```
# We define a helper function to calculate convolutions. It initializes the
# convolutional layer weights and performs corresponding dimensionality
# elevations and reductions on the input and output
def comp_conv2d(conv2d, X):
    # (1, 1) indicates that batch size and the number of channels are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Strip the first two dimensions: examples and channels
    return Y.reshape(Y.shape[2:])

# 1 row and column is padded on either side, so a total of 2 rows or columns
# are added
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

```
# We define a helper function to calculate convolutions. It initializes
# the convolutional layer weights and performs corresponding dimensionality
```

(continues on next page)

(continued from previous page)

```
# elevations and reductions on the input and output
def comp_conv2d(conv2d, X):
    conv2d.initialize()
    # (1, 1) indicates that batch size and the number of channels are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Strip the first two dimensions: examples and channels
    return Y.reshape(Y.shape[2:])

# 1 row and column is padded on either side, so a total of 2 rows or columns␣
↪are added
conv2d = nn.Conv2D(1, kernel_size=3, padding=1)
X = np.random.uniform(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
[07:21:49] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(8, 8)
```

```
# We define a helper function to calculate convolutions. It initializes
# the convolutional layer weights and performs corresponding dimensionality
# elevations and reductions on the input and output
def comp_conv2d(conv2d, X):
    # (1, X.shape, 1) indicates that batch size and the number of channels are␣
↪both 1
    key = jax.random.PRNGKey(d2l.get_seed())
    X = X.reshape((1,) + X.shape + (1,))
    Y, _ = conv2d.init_with_output(key, X)
    # Strip the dimensions: examples and channels
    return Y.reshape(Y.shape[1:3])
# 1 row and column is padded on either side, so a total of 2 rows or columns␣
↪are added
conv2d = nn.Conv(1, kernel_size=(3, 3), padding='SAME')
X = jax.random.uniform(jax.random.PRNGKey(d2l.get_seed()), shape=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

```
# We define a helper function to calculate convolutions. It initializes
# the convolutional layer weights and performs corresponding dimensionality
# elevations and reductions on the input and output
def comp_conv2d(conv2d, X):
    # (1, 1) indicates that batch size and the number of channels are both 1
    X = tf.reshape(X, (1, ) + X.shape + (1, ))
    Y = conv2d(X)
    # Strip the first two dimensions: examples and channels
    return tf.reshape(Y, Y.shape[1:3])
# 1 row and column is padded on either side, so a total of 2 rows or columns
```

(continues on next page)

(continued from previous page)

```
# are added
conv2d = tf.keras.layers.Conv2D(1, kernel_size=3, padding='same')
X = tf.random.uniform(shape=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
TensorShape([8, 8])
```

畳み込みカーネルの高さと幅が異なる場合でも、高さと幅で異なるパディング数を設定することで、出力と入力の高さと幅を同じにできる。

```
# We use a convolution kernel with height 5 and width 3. The padding on either
# side of the height and width are 2 and 1, respectively
conv2d = nn.LazyConv2d(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

```
# We use a convolution kernel with height 5 and width 3. The padding on
# either side of the height and width are 2 and 1, respectively
conv2d = nn.Conv2D(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

```
# We use a convolution kernel with height 5 and width 3. The padding on
# either side of the height and width are 2 and 1, respectively
conv2d = nn.Conv(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

```
# We use a convolution kernel with height 5 and width 3. The padding on
# either side of the height and width are 2 and 1, respectively
conv2d = tf.keras.layers.Conv2D(1, kernel_size=(5, 3), padding='same')
comp_conv2d(conv2d, X).shape
```

```
TensorShape([8, 8])
```

10.3.2 ストライド

相互相関を計算するとき、まず畳み込みウィンドウを入力テンソルの左上隅に置き、その後、下方向と右方向の両方へすべての位置に滑らせていく。これまでの例では、1 要素ずつ移動するのが既定であった。しかし、計算効率のためで

あったり、ダウンサンプリングしたかったりする場合には、ウィンドウを 1 要素より多く移動させて、途中の位置を飛ばすことがある。これは、畳み込みカーネルが大きい場合に特に有用である。なぜなら、基になる画像の広い領域を捉えられるからである。

1 回の移動で進む行数と列数を ストライドと呼ぶ。ここまで、高さ方向と幅方向のストライドはいずれも 1 を使ってきた。ときには、より大きなストライドを使いたいこともある。Fig. 10.3.3 は、高さ方向に 3、幅方向に 2 のストライドをもつ 2 次元相互相関演算を示している。網掛け部分は、出力要素と、その出力計算に使われる入力テンソルおよびカーネルテンソルの要素を表している: $0\times0+0\times1+1\times2+2\times3=8, 0\times0+6\times1+0\times2+0\times3=6$。最初の列の 2 番目の要素が生成されるとき、畳み込みウィンドウは 3 行下に移動していることが分かる。最初の行の 2 番目の要素が生成されるとき、畳み込みウィンドウは 2 列右に移動している。畳み込みウィンドウが入力上をさらに 2 列右へ移動し続けると、入力要素がウィンドウを満たせないため出力は得られない（別の列のパディングを追加しない限り）。

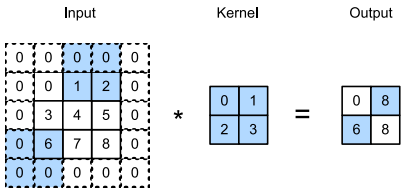


Fig. 10.3.3 高さと幅のストライドがそれぞれ 3 と 2 の相互相関。

一般に、高さ方向のストライドが s_h、幅方向のストライドが s_w のとき、出力の形状は

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor. \tag{10.3.2}$$

となる。

$p_h = k_h - 1$ および $p_w = k_w - 1$ と設定すると、出力形状は $\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$ に簡略化できる。さらに進めて、入力の高さと幅が高さ方向と幅方向のストライドで割り切れるなら、出力形状は $(n_h/s_h) \times (n_w/s_w)$ になる。

以下では、高さ方向と幅方向のストライドをともに 2 に設定し、入力の高さと幅を半分にする。

```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([4, 4])
```

```
conv2d = nn.Conv2D(1, kernel_size=3, padding=1, strides=2)
comp_conv2d(conv2d, X).shape
```

```
(4, 4)
```

```
conv2d = nn.Conv(1, kernel_size=(3, 3), padding=1, strides=2)
comp_conv2d(conv2d, X).shape
```

```
(4, 4)
```

```
conv2d = tf.keras.layers.Conv2D(1, kernel_size=3, padding='same', strides=2)
comp_conv2d(conv2d, X).shape
```

```
TensorShape([4, 4])
```

もう少し複雑な例を見てみよう。

```
conv2d = nn.LazyConv2d(1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([2, 2])
```

```
conv2d = nn.Conv2D(1, kernel_size=(3, 5), padding=(0, 1), strides=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
(2, 2)
```

```
conv2d = nn.Conv(1, kernel_size=(3, 5), padding=(0, 1), strides=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
(2, 2)
```

```
conv2d = tf.keras.layers.Conv2D(1, kernel_size=(3,5), padding='valid',
                                strides=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
TensorShape([2, 1])
```

10.3.3 要約と考察

パディングは出力の高さと幅を増やすことができる。これは、出力が不必要に縮小するのを避けるために、出力の高さと幅を入力と同じにする目的でよく使われる。さらに、すべての画素が同じ頻度で使われることも保証する。通常

は、入力の高さと幅の両側に対称なパディングを選ぶ。この場合、(p_h, p_w) パディングと呼ぶ。最も一般的には $p_h = p_w$ とし、その場合は単にパディング p を選ぶと言う。

ストライドにも同様の慣習がある。水平方向のストライド s_h と垂直方向のストライド s_w が一致するとき、単にストライド s と言う。ストライドは出力の解像度を下げることができ、たとえば $n > 1$ のとき、出力の高さと幅を入力の高さと幅の $1/n$ にまで減らすことがある。既定では、パディングは 0、ストライドは 1 である。

ここまでで扱ったパディングはすべて、単に画像をゼロで拡張するものであった。これは実現が非常に簡単なので、計算上大きな利点がある。さらに、追加のメモリを割り当てることなく、このパディングを暗黙的に利用するよう演算子を設計できる。同時に、CNN が画像内の暗黙的な位置情報を符号化することも可能にする。つまり、「空白」がどこにあるかを学習するだけでよいのである。ゼロパディング以外にも多くの代替手法がある。Alsallakh *et al.* (2020) はそれらについて包括的な概観を与えている（ただし、アーティファクトが生じる場合を除いて、非ゼロパディングをいつ使うべきかについて明確な指針は示していない）。

10.3.4 演習

1. この節の最後のコード例で、カーネルサイズが $(3, 5)$、パディングが $(0, 1)$、ストライドが $(3, 4)$ のとき、出力形状を計算して実験結果と一致するか確認せよ。
2. 音声信号において、ストライド 2 は何に対応するか？
3. ミラーリングパディング、すなわち境界値をそのまま鏡映してテンソルを拡張するパディングを実装せよ。
4. 1 より大きいストライドの計算上の利点は何か？
5. 1 より大きいストライドには、統計的にどのような利点がありうるか？
6. ストライド $\frac{1}{2}$ はどのように実装するか？それは何に対応するか？どのような場合に有用か？

10.4 複数入力チャネルと複数出力チャネル

Section 10.1.4 では、各画像を構成する複数のチャネル（たとえば、カラー画像には赤・緑・青の量を示す標準的な RGB チャネルがある）や、複数チャネルに対する畳み込み層について説明したが、これまでの数値例では、単一の入力チャネルと単一の出力チャネルだけを用いることで、すべてを単純化していた。これにより、入力、畳み込みカーネル、出力をそれぞれ二次元テンソルとして考えることができた。

チャネルを導入すると、入力と隠れ表現の両方が三次元テンソルになる。たとえば、RGB 入力画像の形状は $3 \times h \times w$ である。サイズ 3 のこの軸を チャネル次元と呼ぶ。チャネルという概念は CNN と同じくらい古く、たとえば LeNet-5 (LeCun *et al.*, 1995) でも使われている。この節では、複数入力チャネルと複数出力チャネルをもつ畳み込みカーネルについて、より詳しく見ていく。

```
from d2l import torch as d2l
import torch
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()
```

```
from d2l import jax as d2l
import jax
from jax import numpy as jnp
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

10.4.1 複数入力チャネル

入力データに複数のチャネルが含まれる場合、入力データと同じ数の入力チャネルをもつ畳み込みカーネルを構成する必要がある。そうすることで、入力データとの相互相関を実行できる。入力データのチャネル数を c_i とすると、畳み込みカーネルの入力チャネル数も c_i である必要がある。畳み込みカーネルのウィンドウ形状が $k_\mathrm{h} \times k_\mathrm{w}$ であるとすると、$c_\mathrm{i} = 1$ のときは、畳み込みカーネルを形状 $k_\mathrm{h} \times k_\mathrm{w}$ の二次元テンソルとして考えれば十分である。

しかし、$c_\mathrm{i} > 1$ のときは、各入力チャネルに対して形状 $k_\mathrm{h} \times k_\mathrm{w}$ のテンソルを含むカーネルが必要である。これら c_i 個のテンソルを連結すると、形状 $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$ の畳み込みカーネルが得られる。入力と畳み込みカーネルはいずれも c_i 個のチャネルをもつので、入力の二次元テンソルと畳み込みカーネルの二次元テンソルの間で、各チャネルごとに相互相関を計算し、その c_i 個の結果を加算（チャネル方向に和を取る）して、二次元テンソルを得る。これが、多チャネル入力と複数入力チャネルの畳み込みカーネルとの間の二次元相互相関の結果である。

Fig. 10.4.1 は、2 つの入力チャネルをもつ二次元相互相関の例を示している。網掛け部分は、最初の出力要素と、その出力計算に使われる入力およびカーネルテンソル要素である。$(1\times1+2\times2+4\times3+5\times4)+(0\times0+1\times1+3\times2+4\times3) = 56$。

ここで何が起きているのかを本当に理解するために、私たち自身で複数入力チ

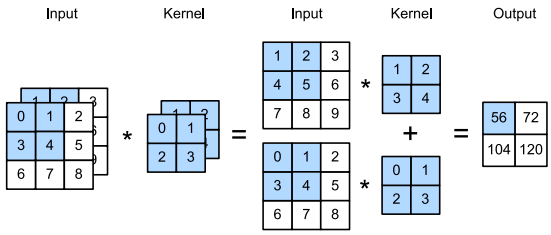


Fig. 10.4.1 2 つの入力チャネルを用いた相互相関の計算。

ャネルをもつ相互相関演算を実装してみよう。やっていることは、チャネルごとに相互相関を行い、その結果を足し合わせているだけだという点に注意してほしい。

```
def corr2d_multi_in(X, K):
    # まず K の 0 次元目（チャネル）を反復し、その後でそれらを足し合わせる
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

```
def corr2d_multi_in(X, K):
    # まず K の 0 次元目（チャネル）を反復し、その後でそれらを足し合わせる
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

```
def corr2d_multi_in(X, K):
    # まず K の 0 次元目（チャネル）を反復し、その後でそれらを足し合わせる
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

```
def corr2d_multi_in(X, K):
    # まず K の 0 次元目（チャネル）を反復し、その後でそれらを足し合わせる
    return tf.reduce_sum([d2l.corr2d(x, k) for x, k in zip(X, K)], axis=0)
```

Fig. 10.4.1 の値に対応する入力テンソル `X` とカーネルテンソル `K` を構成して、相互相関演算の出力を検証できる。

```
X = d2l.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
                [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = d2l.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

```
tensor([[ 56.,  72.],
        [104., 120.]])
```

10.4.2 複数出力チャネル

入力チャネル数にかかわらず、これまでは常に 1 つの出力チャネルに行き着いていた。しかし、Section 10.1.4 で述べたように、各層に複数のチャネルをもつ

ことは本質的である。最も一般的なニューラルネットワークアーキテクチャでは、ニューラルネットワークを深くするにつれてチャネル次元を増やし、通常はダウンサンプリングによって空間分解能と引き換えに、より大きな チャネル深さを得る。直感的には、各チャネルは異なる特徴群に反応すると考えられる。実際には、これはもう少し複雑である。素朴な解釈では、表現はピクセルごと、あるいはチャネルごとに独立に学習されるように思えるかもしれない。そうではなく、チャネルは協調して有用になるように最適化される。つまり、単一のチャネルがエッジ検出器に対応するというより、チャネル空間のある方向がエッジ検出に対応している、と考えるほうがよいのである。

入力チャネル数と出力チャネル数をそれぞれ c_i と c_o、カーネルの高さと幅を k_h と k_w とする。複数の出力チャネルをもつ出力を得るには、各出力チャネルに対して形状 $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$ のカーネルテンソルを作成できる。それらを出力チャネル次元に沿って連結すると、畳み込みカーネルの形状は $c_\mathrm{o} \times c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$ になる。相互相関演算では、各出力チャネルの結果は、その出力チャネルに対応する畳み込みカーネルから計算され、入力テンソルのすべてのチャネルから入力を受け取る。

以下に示すように、複数チャネルの出力を計算する相互相関関数を実装する。

```
def corr2d_multi_in_out(X, K):
    # K の 0 次元目を反復し、そのたびに入力 X との
    # 相互相関演算を行う。すべての結果を
    # まとめてスタックする
    return d2l.stack([corr2d_multi_in(X, k) for k in K], 0)
```

K のカーネルテンソルに K+1 と K+2 を連結することで、3 つの出力チャネルをもつ単純な畳み込みカーネルを構成する。

```
K = d2l.stack((K, K + 1, K + 2), 0)
K.shape
```

```
torch.Size([3, 2, 2, 2])
```

以下では、入力テンソル X に対してカーネルテンソル K を用いて相互相関演算を行う。これで出力には 3 つのチャネルが含まれる。最初のチャネルの結果は、前の入力テンソル X と複数入力チャネル・単一出力チャネルのカーネルによる結果と一致する。

```
corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56.,  72.],
         [104., 120.]],

        [[ 76., 100.],
```

(continues on next page)

(continued from previous page)

```
        [148., 172.]],

       [[ 96., 128.],
        [192., 224.]]])
```

10.4.3 1×1 畳み込み層

最初は、1×1 畳み込み、すなわち $k_h = k_w = 1$ は、あまり意味がないように見える。そもそも、畳み込みは隣接するピクセルを相関させるものである。1×1 畳み込みは明らかにそうではない。それにもかかわらず、これは複雑な深層ネットワークの設計にしばしば含まれる人気のある演算である (Lin *et al.*, 2013, Szegedy *et al.*, 2017)。実際に何をしているのかを、もう少し詳しく見てみよう。

最小のウィンドウを使うため、1×1 畳み込みは、高さ方向と幅方向の隣接要素どうしの相互作用から成るパターンを認識するという、より大きな畳み込み層の能力を失う。1×1 畳み込みで行われる計算は、チャネル次元上だけである。

Fig. 10.4.2 は、3 つの入力チャネルと 2 つの出力チャネルをもつ 1×1 畳み込みカーネルを用いた相互相関計算を示している。入力と出力は同じ高さと幅をもつことに注意してほしい。出力の各要素は、入力画像の 同じ位置にある要素の線形結合から導かれる。1×1 畳み込み層は、各ピクセル位置に適用される全結合層であり、対応する c_i 個の入力値を c_o 個の出力値へ変換すると考えることができる。とはいえ、これは依然として畳み込み層なので、重みはピクセル位置をまたいで共有される。したがって、1×1 畳み込み層に必要な重みは $c_o \times c_i$ 個（加えてバイアス）である。また、畳み込み層の後には通常、非線形変換が続くことにも注意してほしい。これにより、1×1 畳み込みを他の畳み込みに単純にまとめてしまうことはできなくなる。

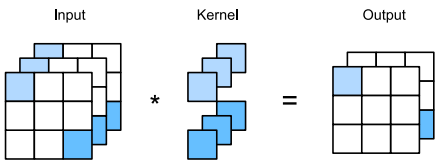


Fig. 10.4.2 相互相関計算は、3 つの入力チャネルと 2 つの出力チャネルをもつ 1×1 畳み込みカーネルを用いる。入力と出力は同じ高さと幅をもつ。

実際にこれが機能するか確認してみよう。全結合層を使って 1×1 畳み込みを実装する。必要なのは、行列積の前後でデータ形状を少し調整することだけである。

```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
```

(continues on next page)

(continued from previous page)

```
    c_o = K.shape[0]
    X = d2l.reshape(X, (c_i, h * w))
    K = d2l.reshape(K, (c_o, c_i))
    # 全結合層での行列積
    Y = d2l.matmul(K, X)
    return d2l.reshape(Y, (c_o, h, w))
```

1×1 畳み込みを行う場合、上の関数は、先に実装した相互相関関数 `corr2d_multi_in_out` と等価である。いくつかのサンプルデータで確認してみよう。

```
X = d2l.normal(0, 1, (3, 3, 3))
K = d2l.normal(0, 1, (2, 3, 1, 1))
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(d2l.reduce_sum(d2l.abs(Y1 - Y2))) < 1e-6
```

```
X = d2l.normal(0, 1, (3, 3, 3))
K = d2l.normal(0, 1, (2, 3, 1, 1))
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(d2l.reduce_sum(d2l.abs(Y1 - Y2))) < 1e-6
```

```
X = jax.random.normal(jax.random.PRNGKey(d2l.get_seed()), (3, 3, 3)) + 0 * 1
K = jax.random.normal(jax.random.PRNGKey(d2l.get_seed()), (2, 3, 1, 1)) + 0 * 1
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(d2l.reduce_sum(d2l.abs(Y1 - Y2))) < 1e-6
```

```
X = d2l.normal((3, 3, 3), 0, 1)
K = d2l.normal((2, 3, 1, 1), 0, 1)
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(d2l.reduce_sum(d2l.abs(Y1 - Y2))) < 1e-6
```

10.4.4 議論

チャネルによって、MLP がもつ大きな非線形性と、特徴の 局所的な解析を可能にする畳み込みの、両方の利点を組み合わせることができる。特にチャネルは、エッジ検出器や形状検出器のような複数の特徴を CNN が同時に扱うことを可能にする。また、平行移動不変性と局所性による劇的なパラメータ削減と、コンピュータビジョンにおける表現力豊かで多様なモデルの必要性との間で、実用的なトレードオフも提供する。

ただし、この柔軟性には代償があることに注意してほしい。サイズ $(h \times w)$ の画像に対して、$k \times k$ 畳み込みを計算するコストは $O(h \cdot w \cdot k^2)$ である。入力チャネル数と出力チャネル数をそれぞれ c_i と c_o とすると、これは $O(h \cdot w \cdot k^2 \cdot c_\mathrm{i} \cdot c_\mathrm{o})$ に増加する。256×256 ピクセルの画像に対して、5×5 カーネルとそれぞれ 128

個の入力チャネルおよび出力チャネルを用いると、計算量は 530 億回を超える（乗算と加算を別々に数えている）。後ほど、たとえばチャネルごとの演算をブロック対角にすることで、ResNeXt (Xie *et al.*, 2017) のようなアーキテクチャにつながる、コスト削減の有効な戦略に出会う。

10.4.5 演習

1. それぞれサイズ k_1 と k_2 の 2 つの畳み込みカーネルがあると仮定する（その間に非線形性はないものとする）。
   1. この演算の結果が 1 つの畳み込みとして表せることを証明せよ。
   2. 等価な 1 つの畳み込みの次元はどれくらいか？
   3. 逆は成り立つか？つまり、任意の畳み込みを常に 2 つのより小さい畳み込みに分解できるか？
2. 形状 $c_\mathrm{i} \times h \times w$ の入力、形状 $c_\mathrm{o} \times c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$ の畳み込みカーネル、パディング $(p_\mathrm{h}, p_\mathrm{w})$、ストライド $(s_\mathrm{h}, s_\mathrm{w})$ があると仮定する。
   1. 順伝播の計算コスト（乗算と加算）はどれくらいか？
   2. メモリ使用量はどれくらいか？
   3. 逆伝播計算のメモリ使用量はどれくらいか？
   4. 逆伝播の計算コストはどれくらいか？
3. 入力チャネル数 c_i と出力チャネル数 c_o の両方を 2 倍にすると、計算回数は何倍になるか？パディングを 2 倍にするとどうなるか？
4. この節の最後の例における変数 `Y1` と `Y2` は完全に同じか？なぜか？
5. 畳み込みウィンドウが 1×1 でない場合でも、畳み込みを行列積として表現せよ。
6. あなたの課題は、$k \times k$ カーネルを用いた高速畳み込みを実装することである。候補となるアルゴリズムの 1 つは、ソース上を水平方向に走査し、幅 k の帯を読み込んで、幅 1 の出力帯を 1 値ずつ計算する方法である。別の方法は、幅 $k + \Delta$ の帯を読み込み、幅 Δ の出力帯を計算することである。なぜ後者のほうが望ましいのか？Δ をどれだけ大きく選ぶべきかに上限はあるか？
7. $c \times c$ の行列があると仮定する。
   1. 行列が b 個のブロックに分割されているとき、ブロック対角行列との積はどれくらい高速になるか？
   2. b 個のブロックをもつことの欠点は何か？少なくとも部分的に、それをどう修正できるか？

10.5 プーリング

多くの場合、最終的なタスクは画像に対する大域的な問いを求める。たとえば、猫が含まれているか？といった具合である。したがって、最終層のユニットは入力全体に敏感であるべきである。情報を徐々に集約し、より粗いマップを得ていくことで、中間層では畳み込み層の利点を保ちながら、最終的には大域的な表現を学習するというこの目標を達成できる。ネットワークを深く進むほど、各隠れノードが敏感になる受容野は（入力に対して相対的に）大きくなる。空間解像度を下げると、畳み込みカーネルがより大きな有効領域を覆うため、この過程が加速される。

さらに、エッジのような低レベル特徴を検出する際には（Section 10.2 で議論したように）、表現がある程度の平行移動不変性を持つことがしばしば望まれる。たとえば、黒と白の境界がはっきりした画像 `X` を取り、画像全体を右に 1 ピクセルずらす、すなわち `Z[i, j] = X[i, j + 1]` とすると、新しい画像 `Z` の出力は大きく異なるかもしれない。エッジは 1 ピクセル分移動しているからである。現実には、物体がまったく同じ場所に現れることはほとんどない。実際、三脚を使い静止した物体を撮影していても、シャッターの動きによるカメラの振動で全体が 1 ピクセル程度ずれることがある（高級カメラにはこの問題に対処するための特別な機能が搭載されている）。

この節では、プーリング層を導入する。これは、畳み込み層の位置に対する感度を緩和することと、表現を空間的にダウンサンプリングすることという二つの目的を果たす。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
import tensorflow as tf
```

(continues on next page)

(continued from previous page)

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = tf.Variable(tf.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w +1)))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j].assign(tf.reduce_max(X[i: i + p_h, j: j + p_w]))
            elif mode =='avg':
                Y[i, j].assign(tf.reduce_mean(X[i: i + p_h, j: j + p_w]))
    return Y
```

10.5.1 最大プーリングと平均プーリング

畳み込み層と同様に、プーリング演算子は固定形状のウィンドウからなり、そのストライドに従って入力のすべての領域上をスライドし、固定形状ウィンドウ（プーリングウィンドウとも呼ばれる）が通過する各位置について 1 つの出力を計算する。しかし、畳み込み層における入力とカーネルの相互相関計算とは異なり、プーリング層にはパラメータがない（カーネルは存在しない）。その代わり、プーリング演算子は決定論的であり、通常はプーリングウィンドウ内の要素の最大値または平均値を計算する。これらの操作はそれぞれ、最大プーリング（略して *max-pooling*）と 平均プーリングと呼ばれる。

平均プーリングは、CNN と同じくらい古くからある手法である。その考え方は画像のダウンサンプリングに似ている。低解像度画像を得るために単に 2 つおき（あるいは 3 つおき）の画素の値を取るのではなく、隣接する画素を平均することで、複数の隣接画素からの情報を統合し、信号対雑音比の高い画像を得ることができる。*max-pooling* は、認知神経科学の文脈で物体認識のために情報集約がどのように階層的に行われうるかを記述する目的で Riesenhuber and Poggio (1999) において導入された。それ以前にも音声認識における先行版があった (Yamaguchi *et al.*, 1990)。ほとんどすべての場合において、max-pooling とも呼ばれるこの手法は、平均プーリングよりも望ましい。

どちらの場合も、相互相関演算子と同様に、プーリングウィンドウは入力テンソルの左上から始まり、左から右へ、上から下へとスライドしていくと考えられる。プーリングウィンドウが各位置に到達するたびに、max か average かに応じて、ウィンドウ内の入力部分テンソルの最大値または平均値を計算する。

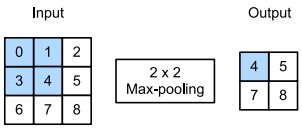


Fig. 10.5.1 プーリングウィンドウの形状が 2×2 の max-pooling。影付き部分は最初の出力要素であり、出力計算に使われる入力テンソル要素でもある: $\max(0, 1, 3, 4) = 4$.

Fig. 10.5.1 の出力テンソルの高さは 2、幅は 2 である。4 つの要素は、それぞれ

のプーリングウィンドウ内の最大値から得られる。

$$\begin{aligned}\max(0,1,3,4) &= 4,\\ \max(1,2,4,5) &= 5,\\ \max(3,4,6,7) &= 7,\\ \max(4,5,7,8) &= 8.\end{aligned} \tag{10.5.1}$$

より一般には、$p \times q$ のプーリング層を、そのサイズの領域上で集約することで定義できる。エッジ検出の問題に戻ると、畳み込み層の出力を 2×2 の max-pooling の入力として用いる。`X` を畳み込み層の入力、`Y` をプーリング層の出力とする。`X[i, j]`、`X[i, j + 1]`、`X[i+1, j]`、`X[i+1, j + 1]` の値が異なるかどうかにかかわらず、プーリング層は常に `Y[i, j] = 1` を出力する。つまり、2×2 の max-pooling 層を使えば、畳み込み層が認識したパターンが高さまたは幅方向に 1 要素以内しか移動しない限り、それを依然として検出できる。

以下のコードでは、`pool2d` 関数でプーリング層の順伝播を実装する。この関数は Section 10.2 の `corr2d` 関数に似ている。ただし、カーネルは不要で、入力の各領域の最大値または平均値として出力を計算する。

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = d2l.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = d2l.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = jnp.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y = Y.at[i, j].set(X[i: i + p_h, j: j + p_w].max())
            elif mode == 'avg':
                Y = Y.at[i, j].set(X[i: i + p_h, j: j + p_w].mean())
    return Y
```

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
<tf.Variable 'Variable:0' shape=(2, 2) dtype=float32, numpy=
array([[4., 5.],
       [7., 8.]], dtype=float32)>
```

Fig. 10.5.1 における入力テンソル `X` を構成して、2 次元 max-pooling 層の出力を検証できる。

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
tensor([[4., 5.],
        [7., 8.]])
```

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
[07:04:06] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
array([[4., 5.],
       [7., 8.]])
```

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
Array([[4., 5.],
       [7., 8.]], dtype=float32)
```

```
pool2d(X, (2, 2), 'avg')
```

```
<tf.Variable 'Variable:0' shape=(2, 2) dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>
```

また、平均プーリング層でも試すことができる。

```
pool2d(X, (2, 2), 'avg')
```

```
tensor([[2., 3.],
        [5., 6.]])
```

```
pool2d(X, (2, 2), 'avg')
```

```
array([[2., 3.],
       [5., 6.]])
```

```
pool2d(X, (2, 2), 'avg')
```

```
Array([[2., 3.],
       [5., 6.]], dtype=float32)
```

```
X = d2l.reshape(d2l.arange(16, dtype=d2l.float32), (1, 4, 4, 1))
X
```

```
<tf.Tensor: shape=(1, 4, 4, 1), dtype=float32, numpy=
array([[[[ 0.],
         [ 1.],
         [ 2.],
         [ 3.]],

        [[ 4.],
         [ 5.],
         [ 6.],
         [ 7.]],

        [[ 8.],
         [ 9.],
         [10.],
         [11.]],

        [[12.],
         [13.],
         [14.],
         [15.]]]], dtype=float32)>
```

10.5.2 パディングとストライド

畳み込み層と同様に、プーリング層も出力形状を変化させる。そしてこれまでと同様に、入力をパディングしストライドを調整することで、望ましい出力形状を得るように演算を調整できる。プーリング層におけるパディングとストライドの使用は、深層学習フレームワークの組み込み 2 次元 max-pooling 層を通して示せる。まず、形状が 4 次元の入力テンソル `X` を構成する。ここで、サンプル数（バッチサイズ）とチャネル数はいずれも 1 である。

```
X = d2l.reshape(d2l.arange(16, dtype=d2l.float32), (1, 1, 4, 4))
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

```
X = d2l.reshape(d2l.arange(16, dtype=d2l.float32), (1, 1, 4, 4))
X
```

```
array([[[[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.]]]])
```

```
X = d2l.reshape(d2l.arange(16, dtype=d2l.float32), (1, 4, 4, 1))
X
```

```
Array([[[[ 0.],
         [ 1.],
         [ 2.],
         [ 3.]],

        [[ 4.],
         [ 5.],
         [ 6.],
         [ 7.]],

        [[ 8.],
         [ 9.],
         [10.],
         [11.]],

        [[12.],
         [13.],
         [14.],
         [15.]]]], dtype=float32)
```

```
pool2d = tf.keras.layers.MaxPool2D(pool_size=[3, 3])
# Pooling has no model parameters, hence it needs no initialization
pool2d(X)
```

```
<tf.Tensor: shape=(1, 1, 1, 1), dtype=float32, numpy=array([[[[10.]]]],␣
↪dtype=float32)>
```

プーリングは領域から情報を集約するため、深層学習フレームワークではプーリングウィンドウサイズとストライドが一致するのが既定である。たとえば、

形状 (3, 3) のプーリングウィンドウを使うと、既定では (3, 3) のストライドになる。

```
pool2d = nn.MaxPool2d(3)
# Pooling has no model parameters, hence it needs no initialization
pool2d(X)
```

```
tensor([[[[10.]]]])
```

```
pool2d = nn.MaxPool2D(3)
# Pooling has no model parameters, hence it needs no initialization
pool2d(X)
```

```
array([[[[10.]]]])
```

```
# Pooling has no model parameters, hence it needs no initialization
nn.max_pool(X, window_shape=(3, 3), strides=(3, 3))
```

```
Array([[[[10.]]]], dtype=float32)
```

```
paddings = tf.constant([[0, 0], [1,0], [1,0], [0,0]])
X_padded = tf.pad(X, paddings, "CONSTANT")
pool2d = tf.keras.layers.MaxPool2D(pool_size=[3, 3], padding='valid',
                                   strides=2)
pool2d(X_padded)
```

```
<tf.Tensor: shape=(1, 2, 2, 1), dtype=float32, numpy=
array([[[[ 5.],
         [ 7.]],

        [[13.],
         [15.]]]], dtype=float32)>
```

言うまでもなく、必要であればストライドとパディングを手動で指定して、フレームワークの既定値を上書きできる。

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

```
pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)
```

```
array([[[[ 5.,  7.],
         [13., 15.]]]])
```

```
X_padded = jnp.pad(X, ((0, 0), (1, 0), (1, 0), (0, 0)), mode='constant')
nn.max_pool(X_padded, window_shape=(3, 3), padding='VALID', strides=(2, 2))
```

```
Array([[[[ 5.],
         [ 7.]],

        [[13.],
         [15.]]]], dtype=float32)
```

```
paddings = tf.constant([[0, 0], [0, 0], [1, 1], [0, 0]])
X_padded = tf.pad(X, paddings, "CONSTANT")

pool2d = tf.keras.layers.MaxPool2D(pool_size=[2, 3], padding='valid',
                                   strides=(2, 3))
pool2d(X_padded)
```

```
<tf.Tensor: shape=(1, 2, 2, 1), dtype=float32, numpy=
array([[[[ 5.],
         [ 7.]],

        [[13.],
         [15.]]]], dtype=float32)>
```

もちろん、以下の例が示すように、任意の高さと幅を持つ任意の長方形のプーリングウィンドウを指定できる。

```
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

```
pool2d = nn.MaxPool2D((2, 3), padding=(0, 1), strides=(2, 3))
pool2d(X)
```

```
array([[[[ 5.,  7.],
         [13., 15.]]]])
```

```
X_padded = jnp.pad(X, ((0, 0), (0, 0), (1, 1), (0, 0)), mode='constant')
nn.max_pool(X_padded, window_shape=(2, 3), strides=(2, 3), padding='VALID')
```

```
Array([[[[ 5.],
         [ 7.]],

        [[13.],
         [15.]]]], dtype=float32)
```

```
# Concatenate along `dim=3` due to channels-last syntax
X = d2l.concat([X, X + 1], 3)
X
```

```
<tf.Tensor: shape=(1, 4, 4, 2), dtype=float32, numpy=
array([[[[ 0.,  1.],
         [ 1.,  2.],
         [ 2.,  3.],
         [ 3.,  4.]],

        [[ 4.,  5.],
         [ 5.,  6.],
         [ 6.,  7.],
         [ 7.,  8.]],

        [[ 8.,  9.],
         [ 9., 10.],
         [10., 11.],
         [11., 12.]],

        [[12., 13.],
         [13., 14.],
         [14., 15.],
         [15., 16.]]]], dtype=float32)>
```

10.5.3 複数チャネル

多チャネル入力データを処理する場合、プーリング層は入力チャネルごとに別々にプーリングし、畳み込み層のようにチャネル方向に入力を合計することはしない。これは、プーリング層の出力チャネル数が入力チャネル数と同じであることを意味する。以下では、テンソル `X` と `X + 1` をチャネル次元で連結し、2チャネルの入力を構成する。

```
X = d2l.concat((X, X + 1), 1)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
```

(continues on next page)

(continued from previous page)

```
         [ 9., 10., 11., 12.],
         [13., 14., 15., 16.]]])
```

```
X = d2l.concat((X, X + 1), 1)
X
```

```
array([[[[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.]],

        [[ 1.,  2.,  3.,  4.],
         [ 5.,  6.,  7.,  8.],
         [ 9., 10., 11., 12.],
         [13., 14., 15., 16.]]])
```

```
# Concatenate along `dim=3` due to channels-last syntax
X = d2l.concat([X, X + 1], 3)
X
```

```
Array([[[[ 0.,  1.],
         [ 1.,  2.],
         [ 2.,  3.],
         [ 3.,  4.]],

        [[ 4.,  5.],
         [ 5.,  6.],
         [ 6.,  7.],
         [ 7.,  8.]],

        [[ 8.,  9.],
         [ 9., 10.],
         [10., 11.],
         [11., 12.]],

        [[12., 13.],
         [13., 14.],
         [14., 15.],
         [15., 16.]]], dtype=float32)
```

```
paddings = tf.constant([[0, 0], [1,0], [1,0], [0,0]])
X_padded = tf.pad(X, paddings, "CONSTANT")
pool2d = tf.keras.layers.MaxPool2D(pool_size=[3, 3], padding='valid',
                                   strides=2)
pool2d(X_padded)
```

```
<tf.Tensor: shape=(1, 2, 2, 2), dtype=float32, numpy=
array([[[[ 5.,  6.],
```

(continues on next page)

(continued from previous page)

```
         [ 7.,  8.]],

        [[13., 14.],
         [15., 16.]]], dtype=float32)>
```

見てわかるように、プーリング後も出力チャネル数は依然として 2 である。

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]],

         [[ 6.,  8.],
          [14., 16.]]]])
```

```
pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)
```

```
array([[[[ 5.,  7.],
         [13., 15.]],

        [[ 6.,  8.],
         [14., 16.]]]])
```

```
X_padded = jnp.pad(X, ((0, 0), (1, 0), (1, 0), (0, 0)), mode='constant')
nn.max_pool(X_padded, window_shape=(3, 3), padding='VALID', strides=(2, 2))
```

```
Array([[[[ 5.,  6.],
         [ 7.,  8.]],

        [[13., 14.],
         [15., 16.]]]], dtype=float32)
```

10.5.4 まとめ

プーリングは非常に単純な操作である。その名の通り、値のウィンドウ上で結果を集約する。ストライドやパディングなど、畳み込みに関する意味論はこれまでと同様に適用される。プーリングはチャネルに依存しない、すなわちチャネル数を変えず、各チャネルに別々に適用されることに注意してほしい。最後に、2 つの代表的なプーリング手法のうち、max-pooling は平均プーリングよりも望ましく、出力にある程度の不変性を与える。よく使われる選択として、出力の空間解像度を 4 分の 1 にするために 2×2 のプーリングウィンドウを選ぶ方法がある。

プーリング以外にも解像度を下げる方法は数多くある。たとえば、stochastic pooling (Zeiler and Fergus, 2013) や fractional max-pooling (Graham, 2014) では、集約にランダム化が組み合わされている。これにより、場合によっては精度がわずかに向上する。最後に、後で注意機構で見るように、出力を集約するより洗練された方法もある。たとえば、クエリと表現ベクトルの整列を用いる方法である。

10.5.5 演習

1. 畳み込みを通して平均プーリングを実装せよ。
2. max-pooling は畳み込みだけでは実装できないことを証明せよ。
3. max-pooling は ReLU 演算、すなわち $\mathrm{ReLU}(x) = \max(0, x)$ を用いて実現できる。
   1. ReLU 演算だけを用いて $\max(a, b)$ を表せ。
   2. これを用いて、畳み込みと ReLU 層によって max-pooling を実装せよ。
   3. 2×2 の畳み込みには何チャネル、何層必要か？3×3 の畳み込みではどうか？
4. プーリング層の計算コストはいくらか？プーリング層への入力サイズが $c \times h \times w$、プーリングウィンドウの形状が $p_\mathrm{h} \times p_\mathrm{w}$、パディングが $(p_\mathrm{h}, p_\mathrm{w})$、ストライドが $(s_\mathrm{h}, s_\mathrm{w})$ であると仮定せよ。
5. max-pooling と average pooling が異なる働きをすると予想するのはなぜか？
6. 別個の最小プーリング層は必要か？別の演算で置き換えられるか？
7. プーリングに softmax 演算を使うこともできる。なぜそれほど一般的でないのだろうか？

10.6 畳み込みニューラルネットワーク（LeNet）

これで、完全に機能する CNN を組み立てるために必要な材料はすべてそろった。以前、画像データに初めて触れたときには、Fashion-MNIST データセットの衣服画像に対して、ソフトマックス回帰による線形モデル (Section 7.4) と MLP (Section 8.2) を適用した。このようなデータを扱いやすくするために、まず各画像を 28×28 の行列から固定長の 784 次元ベクトルへと平坦化し、その後、全結合層で処理した。いまや畳み込み層を扱えるようになったので、画像の空間構造を保持したまま処理できる。さらに、全結合層を畳み込み層に置き換えることで、必要なパラメータ数がはるかに少ない、より簡潔なモデルを得られる。

この節では、*LeNet* を紹介する。LeNet は、コンピュータビジョンタスクでの

性能によって広く注目を集めた、最初期の公開 CNN の一つである。このモデルは、当時 AT&T Bell Labs の研究者だった Yann LeCun によって、画像中の手書き数字を認識する目的で提案され、その名も彼にちなんでいる (LeCun *et al.*, 1998)。この研究は、技術開発における 10 年にわたる研究の集大成であった。LeCun のチームは、バックプロパゲーションによって CNN を訓練することに初めて成功した研究を発表した (LeCun *et al.*, 1989)。

当時、LeNet はサポートベクターマシンに匹敵する優れた結果を達成し、教師あり学習における支配的な手法であったそれに対して、1 桁あたり 1% 未満の誤り率を実現した。LeNet はその後、ATM 機での入金処理のために数字認識へと応用された。今日に至るまで、1990 年代に Yann LeCun と同僚の Leon Bottou が書いたコードをそのまま動かしている ATM もある！

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
from types import FunctionType
```

```
import tensorflow as tf
from d2l import tensorflow as d2l
```

10.6.1 LeNet

大まかに言うと、LeNet（LeNet-5）は 2 つの部分から構成される。(i) 2 つの畳み込み層からなる畳み込みエンコーダと、(ii) 3 つの全結合層からなる密なブロックである。アーキテクチャは Fig. 10.6.1 に要約されている。

各畳み込みブロックの基本単位は、畳み込み層、シグモイド活性化関数、そしてその後に続く平均プーリング操作である。ReLU と最大プーリングのほうが性能は良いのだが、当時はまだ発見されていなかった。各畳み込み層は 5×5 のカーネルとシグモイド活性化関数を使う。これらの層は、空間的に配置された入力を複数の 2 次元特徴マップへと写像し、通常はチャネル数を増やす。最初の畳み込み層の出力チャネル数は 6、2 番目は 16 である。各 2×2 のプーリング操作（ストライド 2）は、空間的なダウンサンプリングによって次元を 4 分の 1 に削減する。畳み込みブロックの出力形状は（バッチサイズ、チャネル数、高さ、幅）で与えられる。

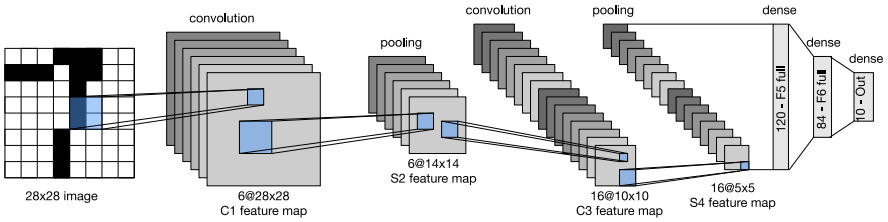


Fig. 10.6.1 LeNet におけるデータの流れ。入力は手書き数字で、出力は 10 個の候補に対する確率である。

畳み込みブロックの出力を密なブロックへ渡すためには、ミニバッチ内の各サンプルを平坦化する必要がある。言い換えると、この 4 次元入力を、全結合層が期待する 2 次元入力へ変換する。念のため言うと、ここで欲しい 2 次元表現では、第 1 次元がミニバッチ内のサンプルを表し、第 2 次元が各サンプルの平坦なベクトル表現を表す。LeNet の密なブロックは 3 つの全結合層からなり、それぞれの出力数は 120、84、10 である。なお、これは分類を行っているので、10 次元の出力層は可能な出力クラス数に対応する。

LeNet の内部で何が起きているのかを本当に理解するところまで来るには少し手間がかかるかもしれないが、次のコード片を見れば、このようなモデルを現代の深層学習フレームワークで実装するのが驚くほど簡単だと納得できるはずである。必要なのは `Sequential` ブロックをインスタンス化し、Section 8.4.2 で紹介した Xavier 初期化を用いて、適切な層を順につなげるだけである。

```
def init_cnn(module):  #@save
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)
```

```
class LeNet(d2l.Classifier):  #@save
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nn.Conv2D(channels=6, kernel_size=5, padding=2,
                          activation='sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Conv2D(channels=16, kernel_size=5, activation='sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Dense(120, activation='sigmoid'),
                nn.Dense(84, activation='sigmoid'),
```

(continues on next page)

(continued from previous page)

```
                nn.Dense(num_classes))
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Flatten(),
                nn.LazyLinear(120), nn.Sigmoid(),
                nn.LazyLinear(84), nn.Sigmoid(),
                nn.LazyLinear(num_classes))
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=6, kernel_size=5,
                                       activation='sigmoid', padding='same'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Conv2D(filters=16, kernel_size=5,
                                       activation='sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Flatten(),
                tf.keras.layers.Dense(120, activation='sigmoid'),
                tf.keras.layers.Dense(84, activation='sigmoid'),
                tf.keras.layers.Dense(num_classes)])
```

```
class LeNet(d2l.Classifier):  #@save
    """The LeNet-5 model."""
    lr: float = 0.1
    num_classes: int = 10
    kernel_init: FunctionType = nn.initializers.xavier_uniform

    def setup(self):
        self.net = nn.Sequential([
            nn.Conv(features=6, kernel_size=(5, 5), padding='SAME',
                    kernel_init=self.kernel_init()),
            nn.sigmoid,
            lambda x: nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2)),
            nn.Conv(features=16, kernel_size=(5, 5), padding='VALID',
                    kernel_init=self.kernel_init()),
            nn.sigmoid,
            lambda x: nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2)),
            lambda x: x.reshape((x.shape[0], -1)),  # flatten
            nn.Dense(features=120, kernel_init=self.kernel_init()),
            nn.sigmoid,
            nn.Dense(features=84, kernel_init=self.kernel_init()),
            nn.sigmoid,
            nn.Dense(features=self.num_classes, kernel_init=self.kernel_init())
        ])
```

```
class LeNet(d2l.Classifier):  #@save
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nn.Conv2D(channels=6, kernel_size=5, padding=2,
                          activation='sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Conv2D(channels=16, kernel_size=5, activation='sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Dense(120, activation='sigmoid'),
                nn.Dense(84, activation='sigmoid'),
                nn.Dense(num_classes))
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Flatten(),
                nn.LazyLinear(120), nn.Sigmoid(),
                nn.LazyLinear(84), nn.Sigmoid(),
                nn.LazyLinear(num_classes))
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=6, kernel_size=5,
                                       activation='sigmoid', padding='same'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Conv2D(filters=16, kernel_size=5,
                                       activation='sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Flatten(),
                tf.keras.layers.Dense(120, activation='sigmoid'),
                tf.keras.layers.Dense(84, activation='sigmoid'),
                tf.keras.layers.Dense(num_classes)])
```

```
class LeNet(d2l.Classifier):  #@save
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nn.Conv2D(channels=6, kernel_size=5, padding=2,
                          activation='sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Conv2D(channels=16, kernel_size=5, activation='sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Dense(120, activation='sigmoid'),
                nn.Dense(84, activation='sigmoid'),
                nn.Dense(num_classes))
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
```

(continues on next page)

(continued from previous page)

```
            self.net = nn.Sequential(
                nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Flatten(),
                nn.LazyLinear(120), nn.Sigmoid(),
                nn.LazyLinear(84), nn.Sigmoid(),
                nn.LazyLinear(num_classes))
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=6, kernel_size=5,
                                       activation='sigmoid', padding='same'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Conv2D(filters=16, kernel_size=5,
                                       activation='sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Flatten(),
                tf.keras.layers.Dense(120, activation='sigmoid'),
                tf.keras.layers.Dense(84, activation='sigmoid'),
                tf.keras.layers.Dense(num_classes)])
```

```
@d2l.add_to_class(d2l.Classifier)  #@save
def layer_summary(self, X_shape):
    X = d2l.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)

model = LeNet()
model.layer_summary((1, 1, 28, 28))
```

```
Conv2D output shape:         (1, 6, 28, 28)
AvgPool2D output shape:      (1, 6, 14, 14)
Conv2D output shape:         (1, 16, 10, 10)
AvgPool2D output shape:      (1, 16, 5, 5)
Dense output shape:  (1, 120)
Dense output shape:  (1, 84)
Dense output shape:  (1, 10)
[07:35:41] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
@d2l.add_to_class(d2l.Classifier)  #@save
def layer_summary(self, X_shape, key=d2l.get_key()):
    X = jnp.zeros(X_shape)
    params = self.init(key, X)
    bound_model = self.clone().bind(params, mutable=['batch_stats'])
    _ = bound_model(X)
    for layer in bound_model.net.layers:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)
```

(continues on next page)

(continued from previous page)

```
model = LeNet()
model.layer_summary((1, 28, 28, 1))
```

```
Conv output shape:   (1, 28, 28, 6)
PjitFunction output shape:   (1, 28, 28, 6)
function output shape:       (1, 14, 14, 6)
Conv output shape:   (1, 10, 10, 16)
PjitFunction output shape:   (1, 10, 10, 16)
function output shape:       (1, 5, 5, 16)
function output shape:       (1, 400)
Dense output shape:  (1, 120)
PjitFunction output shape:   (1, 120)
Dense output shape:  (1, 84)
PjitFunction output shape:   (1, 84)
Dense output shape:  (1, 10)
```

```
@d2l.add_to_class(d2l.Classifier)  #@save
def layer_summary(self, X_shape):
    X = d2l.normal(X_shape)
    for layer in self.net.layers:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)

model = LeNet()
model.layer_summary((1, 28, 28, 1))
```

```
Conv2D output shape:         (1, 28, 28, 6)
AveragePooling2D output shape:       (1, 14, 14, 6)
Conv2D output shape:         (1, 10, 10, 16)
AveragePooling2D output shape:       (1, 5, 5, 16)
Flatten output shape:        (1, 400)
Dense output shape:  (1, 120)
Dense output shape:  (1, 84)
Dense output shape:  (1, 10)
```

ここでは LeNet の再現にあたり少し手を加え、ガウス活性化層をソフトマックス層に置き換えている。ガウスデコーダは現在ではほとんど使われないため、これにより実装が大幅に単純化される。それ以外は、このネットワークは元の LeNet-5 アーキテクチャと一致している。

ネットワークの内部で何が起きているか見てみよう。単一チャネル（白黒）の 28×28 画像をネットワークに通し、各層での出力形状を表示することで、Fig. 10.6.2 で期待される動作と整合していることを確認するために、モデルを調べることができる。

```
@d2l.add_to_class(d2l.Classifier)  #@save
def layer_summary(self, X_shape):
```

(continues on next page)

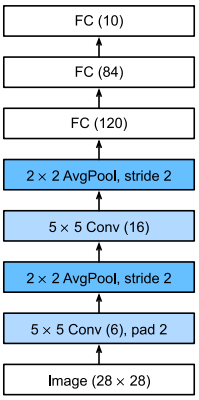


Fig. 10.6.2 LeNet-5 の圧縮表記。

(continued from previous page)

```
    X = d2l.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)

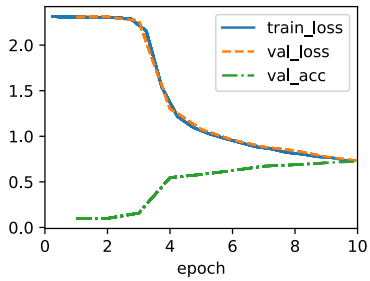
model = LeNet()
model.layer_summary((1, 1, 28, 28))
```

```
Conv2d output shape:         torch.Size([1, 6, 28, 28])
Sigmoid output shape:        torch.Size([1, 6, 28, 28])
AvgPool2d output shape:      torch.Size([1, 6, 14, 14])
Conv2d output shape:         torch.Size([1, 16, 10, 10])
Sigmoid output shape:        torch.Size([1, 16, 10, 10])
AvgPool2d output shape:      torch.Size([1, 16, 5, 5])
Flatten output shape:        torch.Size([1, 400])
Linear output shape:         torch.Size([1, 120])
Sigmoid output shape:        torch.Size([1, 120])
Linear output shape:         torch.Size([1, 84])
Sigmoid output shape:        torch.Size([1, 84])
Linear output shape:         torch.Size([1, 10])
```

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=0.1)
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```

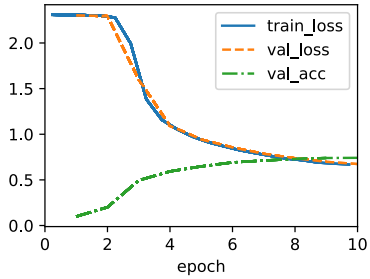
```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=0.1)
if tab.selected('pytorch'):
```

(continues on next page)

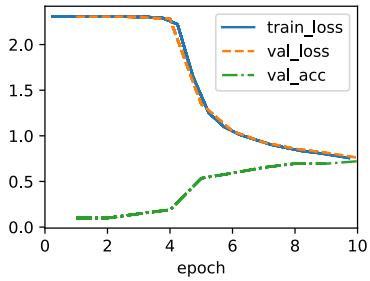


(continued from previous page)

```
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```




```
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=128)
with d2l.try_gpu():
    model = LeNet(lr=0.1)
    trainer.fit(model, data)
```



畳み込みブロック全体を通して、各層における表現の高さと幅は（前の層と比べて）減少していることに注意してほしい。最初の畳み込み層では、5×5 カーネルを使ったときに生じる高さと幅の減少を補うために、2 ピクセルのパディングを使っている。余談だが、元の MNIST OCR データセットにおける 28×28 ピクセルという画像サイズは、元の 32×32 ピクセルのスキャン画像から 2 ピクセ

ル分の行（および列）を切り取った結果である。これは主として、当時はメガバイト単位の節約が重要だったため、容量を節約する（30% の削減）ために行われた。

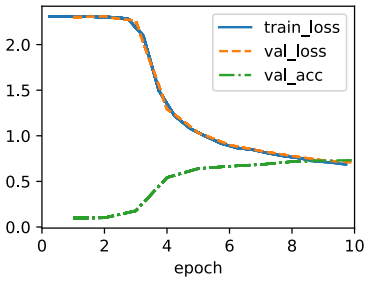
対照的に、2 番目の畳み込み層ではパディングを行わないため、高さと幅はどちらも 4 ピクセルずつ減少する。層を上へ進むにつれて、チャネル数は入力の 1 から、最初の畳み込み層の後に 6、2 番目の畳み込み層の後に 16 へと増加する。しかし、各プーリング層は高さと幅を半分にする。最後に、各全結合層が次元を削減し、最終的にクラス数と一致する次元の出力を生成する。

10.6.2 学習

モデルを実装したので、LeNet-5 モデルが Fashion-MNIST でどの程度うまくいくか実験してみよう。

CNN はパラメータ数が少ない一方で、各パラメータがより多くの乗算に関与するため、同程度の深さの MLP よりも計算コストが高くなることがある。GPU にアクセスできるなら、学習を高速化するためにここで活用するよい機会かもしれない。なお、`d2l.Trainer` クラスが細部をすべて処理してくれる。デフォルトでは、利用可能なデバイス上でモデルパラメータを初期化する。MLP のときと同様に、損失関数はクロスエントロピーであり、ミニバッチ確率的勾配降下法によって最小化する。

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=0.1)
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```



10.6.3 要約

この章では大きな進歩を遂げた。1980 年代の MLP から、1990 年代から 2000 年代初頭の CNN へと移った。たとえば LeNet-5 の形で提案されたアーキテクチャは、今日においてもなお意味を持っている。Fashion-MNIST における誤り率について、LeNet-5 で達成できる性能を、MLP (Section 8.2) で達成可能な最良のものや、ResNet (Section 11.6) のようなはるかに高度なアーキテクチャと比較してみ

る価値がある。LeNet は前者よりも後者にずっと近いものである。後ほど見るように、主な違いの一つは、より多くの計算資源が、はるかに複雑なアーキテクチャを可能にしたことである。

2 つ目の違いは、LeNet を実装することの相対的な容易さである。かつては、C++ とアセンブリコードで何か月もかかる工学的課題であり、SN を改善するためのエンジニアリング、初期の Lisp ベースの深層学習ツール (Bottou and Le Cun, 1988)、そして最終的にはモデルを用いた試行錯誤が必要であったが、今では数分で実現できる。この驚くべき生産性向上こそが、深層学習モデル開発を大きく民主化したのである。次の章では、このウサギの穴をさらに下っていき、どこへたどり着くのかを見ていく。

10.6.4 演習

1. LeNet を現代風にしよう。以下の変更を実装してテストせよ。
    1. 平均プーリングを最大プーリングに置き換える。
    2. ソフトマックス層を ReLU に置き換える。
2. 最大プーリングと ReLU に加えて、LeNet 風ネットワークのサイズを変更して精度を改善してみよう。
    1. 畳み込みウィンドウのサイズを調整する。
    2. 出力チャネル数を調整する。
    3. 畳み込み層の数を調整する。
    4. 全結合層の数を調整する。
    5. 学習率やその他の学習設定（たとえば初期化やエポック数）を調整する。
3. 改良したネットワークを元の MNIST データセットで試してみよう。
4. 異なる入力（たとえばセーターやコート）に対する LeNet の第 1 層と第 2 層の活性化を表示せよ。
5. ネットワークに大きく異なる画像（たとえば猫、車、あるいはランダムノイズ）を入力すると、活性化はどうなるか？

11 現代的な畳み込みニューラルネットワーク

CNN を組み立てる基本を理解した今、現代的な CNN アーキテクチャを見ていこう。この巡回は、次々と登場する刺激的な新設計があまりにも多いため、必然的に不完全なものになる。これらが重要なのは、画像認識タスクに直接使えるだけでなく、追跡 (Zhang *et al.*, 2021)、セグメンテーション (Long *et al.*, 2015)、物体検出 (Redmon and Farhadi, 2018)、スタイル変換 (Gatys *et al.*, 2016) といったより高度なタスクのための基本的な特徴生成器としても機能するからである。本章の多くの節は、ある時点で（あるいは現在も）多くの研究プロジェクトや実運用システムの基盤モデルとなった、重要な CNN アーキテクチャに対応している。これらのネットワークはいずれも一時的に支配的なアーキテクチャとなり、その多くが、2010 年以来コンピュータビジョンにおける教師あり学習の進歩を測る指標となってきた ImageNet コンペティション[75] の勝者または準優勝者であった。CNN に代わって Transformer が台頭し始めたのはごく最近のことで、Dosovitskiy *et al.* (2021) に始まり、Swin Transformer (Liu *et al.*, 2021) がそれに続いた。この発展については、後ほど Section 14 で扱う。

[75] 

深いニューラルネットワークという考え方自体は非常に単純だが（たくさんの層を積み重ねるだけである）、アーキテクチャやハイパーパラメータの選択によって性能は大きく変わり得る。本章で説明するニューラルネットワークは、直感、いくつかの数学的洞察、そして多くの試行錯誤の産物である。これらのモデルは年代順に紹介するが、これは部分的には歴史の流れを伝え、皆さん自身がこの分野がどこへ向かっているのかについて直感を養い、ひいては独自のアーキテクチャを考案できるようにするためである。たとえば、本章で説明するバッチ正規化と残差接続は、深いモデルを学習・設計するための 2 つの人気のある考え方を提供してきたが、どちらもその後、コンピュータビジョン以外のアーキテクチャにも適用されている。

現代的な CNN の巡回は、まず AlexNet (Krizhevsky *et al.*, 2012) から始める。これは、大規模な視覚課題で従来のコンピュータビジョン手法を打ち負かした最初の大規模ネットワークである。次に、いくつもの繰り返しブロックを用いる VGG ネットワーク (Simonyan and Zisserman, 2014)、入力に対してパッチごとにニューラルネットワーク全体を畳み込む network in network (NiN) (Lin *et al.*, 2013)、マルチブランチ畳み込みを用いる GoogLeNet (Szegedy *et al.*, 2015)、コンピュータビジョンで今なお最も人気のある既製アーキテクチャの 1 つである residual network (ResNet) (He *et al.*, 2016)、より疎な接続のための ResNeXt ブロック (Xie *et al.*, 2017)、そして残差アーキテクチャを一般化した DenseNet (Huang *et al.*, 2017) を取り上げる。時が経つにつれて、座標シフト（ShiftNet）(Wu *et al.*, 2018) のような効率的なネットワークのための特別な最適化が数多く開発された。これらは、MobileNet v3 (Howard *et al.*, 2019) のような効率的アーキテクチャの自動探

索へと結実した。また、Radosavovic *et al.* (2020) による半自動的な設計探索も含まれており、これが後ほど本章で議論する RegNetX/Y につながった。この研究は、効率的な設計空間を探すうえで、総当たり計算と実験者の創意工夫を結びつける道筋を示しているという点で示唆に富んでいる。さらに注目すべきなのは Liu *et al.* (2022) の研究で、学習技術（たとえば最適化手法、データ拡張、正則化）が精度向上において決定的な役割を果たすことを示している。また、計算量とデータ量の増加を踏まえると、畳み込み窓の大きさのような長年の前提も見直す必要があることを示している。この点や他にも多くの問いについては、本章を通して順に扱っていく。

11.1 畳み込み深層ニューラルネットワーク（AlexNet）

CNN は、LeNet の導入 (LeCun *et al.*, 1995) 以降、コンピュータビジョンおよび機械学習のコミュニティではよく知られていたが、すぐにこの分野を席巻したわけではない。LeNet は初期の小規模データセットでは良好な結果を示したが、より大規模で現実的なデータセットに対して CNN を訓練する性能と実現可能性は、まだ確立されていなかった。実際、1990 年代初頭から 2012 年の画期的な結果 (Krizhevsky *et al.*, 2012) までの長い期間の多くにおいて、ニューラルネットワークは、カーネル法 (Schölkopf and Smola, 2002)、アンサンブル法 (Freund and Schapire, 1996)、構造化推定 (Taskar *et al.*, 2004) などの他の機械学習手法にしばしば劣っていた。

コンピュータビジョンに関しては、この比較は必ずしも正確ではない。つまり、畳み込みネットワークへの入力は生の、あるいは（たとえば中心化のような）軽い前処理を施した画素値から成るが、実務家が生の画素を従来のモデルにそのまま入力することはない。代わりに、典型的なコンピュータビジョンのパイプラインは、SIFT (Lowe, 2004)、SURF (Bay *et al.*, 2006)、visual words の bag (Sivic and Zisserman, 2003) などの特徴抽出パイプラインを手作業で設計するものであった。特徴を 学習するのではなく、特徴は 作り込まれていた。進歩の大半は、一方では特徴抽出に関するより巧妙なアイデア、他方では幾何学 (Hartley and Zisserman, 2000) に対する深い洞察によってもたらされた。学習アルゴリズムは、しばしば後回しにされていた。

1990 年代にはいくつかのニューラルネットワーク用アクセラレータが利用可能であったが、深い多チャネル・多層の CNN を多数のパラメータで構成するには、まだ十分な性能がなかった。たとえば、1999 年の NVIDIA GeForce 256 は、ゲーム以外の処理に対する意味のあるプログラミング基盤もないまま、加算や乗算などの浮動小数点演算を 1 秒あたり最大 4 億 8 千万回（MFLOPS）しか処理できなかった。今日のアクセラレータは、1 デバイスあたり 1000 TFLOPs を超える性能を発揮できる。さらに、データセットもまだ比較的小規模であった。60,000 枚の低解像度 28×28 ピクセル画像に対する OCR は、非常に難しい課題と

見なされていた。これらの障害に加えて、ニューラルネットワークを訓練するための重要な工夫、すなわちパラメータ初期化のヒューリスティクス (Glorot and Bengio, 2010)、確率的勾配降下法の巧妙な変種 (Kingma and Ba, 2014)、非飽和活性化関数 (Nair and Hinton, 2010)、効果的な正則化手法 (Srivastava *et al.*, 2014) なども、まだ欠けていた。

したがって、エンドツーエンド（画素から分類まで）のシステムを訓練するのではなく、古典的なパイプラインは次のようなものであった。

1. 興味深いデータセットを入手する。初期の頃は、これらのデータセットには高価なセンサーが必要でした。たとえば、1994 年の Apple QuickTake 100[76] は、わずか $1000 で、最大 8 枚の画像を保存できる、驚くべき 0.3 メガピクセル（VGA）解像度を備えていました。
2. 光学、幾何学、その他の解析的手法、そして時には幸運な大学院生による偶然の発見に基づく手作りの特徴を用いてデータセットを前処理する。
3. SIFT（scale-invariant feature transform）(Lowe, 2004)、SURF（speeded up robust features）(Bay *et al.*, 2006)、あるいは他の多くの手調整されたパイプラインのような標準的な特徴抽出器にデータを通す。OpenCV は今日でも SIFT 抽出器を提供している！
4. 得られた表現を、好みの分類器、たいていは線形モデルかカーネル法に投入して分類器を訓練する。

[76] 

機械学習研究者に話を聞けば、彼らは機械学習が重要で美しいものだと答えるだろう。さまざまな分類器の性質は洗練された理論によって証明され (Boucheron *et al.*, 2005)、凸最適化 (Boyd and Vandenberghe, 2004) はそれらを得るための主流となっていた。機械学習分野は活気に満ち、厳密で、きわめて有用であった。しかし、コンピュータビジョン研究者に話を聞けば、まったく異なる話が返ってきただろう。彼らが語る画像認識の不都合な真実は、新しい学習アルゴリズムではなく、特徴、幾何学 (Hartley and Zisserman, 2000, Hartley and Kahl, 2009)、そして工学が進歩を牽引していたということである。コンピュータビジョン研究者は、少し大きい、あるいは少しきれいなデータセット、あるいは少し改善された特徴抽出パイプラインのほうが、どんな学習アルゴリズムよりも最終的な精度にずっと大きく影響すると、もっともな理由で考えていた。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import np, init, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
```

(continues on next page)

(continued from previous page)

```
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

11.1.1 表現学習

状況を別の言い方で表すなら、パイプラインで最も重要な部分は表現であった。そして 2012 年までは、その表現は主として機械的に計算されていた。実際、新しい特徴関数の設計、結果の改善、手法の記述は、論文の中でいずれも大きな位置を占めていた。SIFT (Lowe, 2004)、SURF (Bay *et al.*, 2006)、HOG（histograms of oriented gradient）(Dalal and Triggs, 2005)、visual words の bag (Sivic and Zisserman, 2003)、および同様の特徴抽出器が主流であった。

Yann LeCun、Geoff Hinton、Yoshua Bengio、Andrew Ng、Shun-ichi Amari、Juergen Schmidhuber などを含む別の研究者グループは、異なる構想を持っていた。彼らは、特徴そのものが学習されるべきだと考えていた。さらに、特徴が十分に複雑であるためには、複数の共同学習された層から階層的に構成され、各層が学習可能なパラメータを持つべきだと考えていた。画像の場合、最下層は動物の視覚系が入力を処理する仕組みに倣って、エッジ、色、テクスチャを検出するようになるかもしれない。特に、sparse coding (Olshausen and Field, 1996) によって得られるような視覚特徴の自動設計は、現代の CNN が登場するまで未解決の課題であった。画像データから特徴を自動生成するというアイデアが大きな支持を得たのは、Dean *et al.* (2012), Le (2013) になってからであった。

最初の現代的な CNN (Krizhevsky *et al.*, 2012) は、その発明者の一人である Alex Krizhevsky にちなんで *AlexNet* と名付けられ、LeNet に対する主として進化的な改良でした。これは 2012 年の ImageNet コンペティションで優れた性能を達成した。

興味深いことに、ネットワークの最下層では、モデルは従来のフィルタに似た特徴抽出器を学習した。Fig. 11.1.1 は低レベルの画像記述子を示している。ネットワークのより高い層は、これらの表現を基にして、目、鼻、草の葉などのより大きな構造を表現するかもしれない。さらに高い層では、人、飛行機、犬、フリスビーのような物体全体を表現するかもしれない。最終的には、最後の隠れ状態が画像の内容を要約したコンパクトな表現を学習し、異なるカテゴリに属するデータを容易に分離できるようになる。

AlexNet（2012）とその前身である LeNet（1995）は、多くのアーキテクチャ要素を共有している。では、なぜこれほど長い時間がかかったのだろうか？重要な違いは、過去 20 年間で利用可能なデータ量と計算能力が大幅に増加したことであった。そのため AlexNet ははるかに大規模であり、1995 年に利用可能だった CPU と比べて、より多くのデータで、より高速な GPU 上で訓練された。

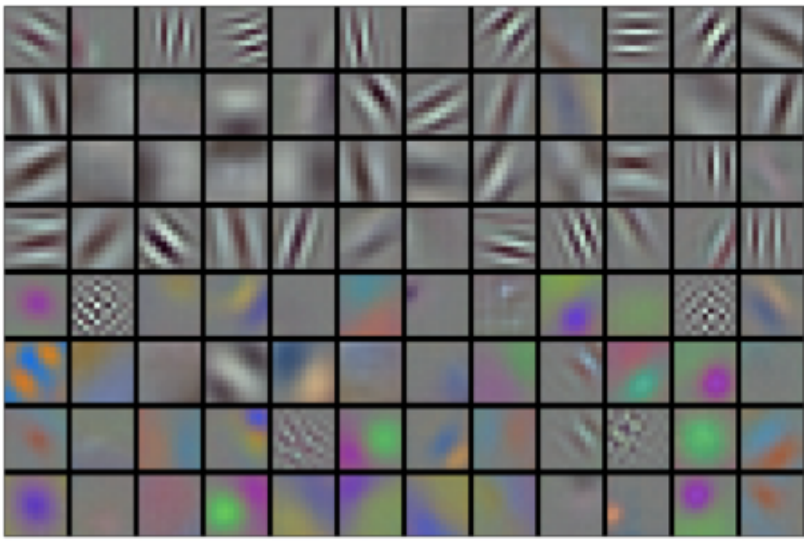


Fig. 11.1.1 AlexNet の第 1 層で学習された画像フィルタ。再現図は Krizhevsky et al. (2012) より。

欠けていた要素: データ

多数の層を持つ深層モデルは、従来手法（たとえば線形法やカーネル法）に基づく凸最適化を大きく上回る段階に入るために、大量のデータを必要とする。しかし、コンピュータのストレージ容量の制約、（画像）センサーの相対的な高コスト、そして 1990 年代における研究予算の比較的厳しさのため、多くの研究は小さなデータセットに依存していた。多くの論文は UCI のデータセット集に依拠しており、その多くは、低解像度で撮影され、しばしば人工的にきれいな背景を持つ、数百枚、あるいは（せいぜい）数千枚の画像しか含んでいなかった。

2009 年、ImageNet データセットが公開され (Deng *et al.*, 2009)、1000 の異なるカテゴリの物体からそれぞれ 1000 例、合計 100 万例からモデルを学習するという課題を研究者に突きつけた。カテゴリ自体は WordNet (Miller, 1995) における最も一般的な名詞ノードに基づいていた。ImageNet チームは Google Image Search を使って各カテゴリの大規模な候補集合を事前にふるい分け、Amazon Mechanical Turk のクラウドソーシング・パイプラインを用いて、各画像が対応するカテゴリに属するかどうかを確認した。この規模は前例がなく、他のデータセットを 1 桁以上上回っていた（たとえば CIFAR-100 は 60,000 枚の画像である）。もう一つの特徴は、画像が 80 million-sized の TinyImages データセット (Torralba *et al.*, 2008) のような 32×32 ピクセルのサムネイルとは異なり、比較的高解像度の 224×224 ピクセルであったことである。これにより、より高次の特徴を形成できた。これに対応するコンペティションである ImageNet Large Scale Visual Recognition Challenge (Russakovsky *et al.*, 2015) は、コンピュータビジョンと機械学習研究を前進させ、研究者に、学術界がそれまで考えていたよりも大きな規模で、どのモデルが最良の性能を示すのかを突き止めるよう挑んだ。今日で

は、1000 万枚の画像を扱うことも前例のないことではない。たとえば、公開されている YFCC-100M データセットには、なんと 1 億枚の画像が含まれている。同時に、医療画像のように特定の視覚タスクでは、十分な高品質のデータを得ることが依然として非常に困難な場合がある。LAION-5B (Schuhmann *et al.*, 2022) のような最大級の視覚データセットには、追加のメタデータを伴う数十億枚の画像が含まれている。

欠けていた要素: ハードウェア

深層学習モデルは計算資源を大量に消費する。訓練には数百エポックかかることがあり、各反復では計算コストの高い線形代数演算を多数の層に通してデータを流す必要がある。これが、1990 年代から 2000 年代初頭にかけて、より効率的に最適化された凸目的関数に基づく単純なアルゴリズムが好まれた主な理由の一つである。

グラフィックス処理装置（GPU）は、深層学習を実用的にするうえでゲームチェンジャーであることが証明された。これらのチップはもともと、コンピュータゲーム向けのグラフィックス処理を高速化するために開発されていた。特に、多くのコンピュータグラフィックス処理に必要な、高スループットの 4×4 行列—ベクトル積に最適化されていた。幸いなことに、その数学は畳み込み層の計算に必要なものと驚くほど似ている。その頃、NVIDIA と ATI は GPU を一般計算向けに最適化し始めており (Fernando, 2004)、それらを *general-purpose GPUs*（GPGPUs）として売り出すまでになっていた。

直感を得るために、現代のマイクロプロセッサ（CPU）のコアを考えてみよう。各コアはかなり高性能で、高いクロック周波数で動作し、大きなキャッシュ（L3 で数メガバイトに達することもある）を備えている。各コアは、分岐予測器、深いパイプライン、専用実行ユニット、投機実行、そしてその他多くの付加機能を備え、幅広い命令を実行するのに適している。これにより、複雑な制御フローを持つ多種多様なプログラムを実行できる。しかし、この見かけ上の強みは、同時に弱点でもある。汎用コアは構築コストが非常に高いのである。制御フローの多い汎用コードには優れている。そのためには、実際に計算が行われる ALU（arithmetic logical unit）だけでなく、前述の付加機能すべて、さらにメモリインターフェース、コア間のキャッシュ制御、高速相互接続などにも多くのチップ面積が必要になる。CPU は、専用ハードウェアと比べると、単一のタスクに関しては相対的に不得手である。現代のノートパソコンは 4–8 コアを備え、ハイエンドサーバーでさえソケットあたり 64 コアを超えることはまれである。単純に、費用対効果がよくないからである。

これに対して、GPU は数千の小さな処理要素から構成されることがある（NIVIDA の最新の Ampere チップは最大 6912 CUDA コアを備えます）。それらはしばしば、より大きなグループ（NVIDIA では warp と呼ぶ）にまとめられる。詳細は NVIDIA、AMD、ARM、その他のチップベンダーで多少異なる。各コアは比較的弱く、約 1GHz のクロック周波数で動作するが、そのようなコアの総数が、GPU を CPU より桁違いに高速にしている。たとえば、NVIDIA の最近の Ampere A100 GPU は、特殊な 16 ビット精度（BFLOAT16）の行列—行

列乗算で 1 チップあたり 300 TFLOPs 超を提供し、より汎用的な浮動小数点演算（FP32）では最大 20 TFLOPs を実現する。一方、CPU の浮動小数点性能は 1 TFLOPs を超えることはめったにない。たとえば、Amazon の Graviton 3 は 16 ビット精度演算でピーク 2 TFLOPs に達し、これは Apple の M1 プロセッサの GPU 性能に近い値である。

GPU が FLOPs の観点で CPU よりはるかに高速である理由は多くある。第一に、消費電力はクロック周波数に対して 二乗的に増加する傾向がある。したがって、4 倍高速に動作する CPU コア 1 個の電力予算で（典型的な値です）、速度を $\frac{1}{4}$ に落とした GPU コアを 16 個使えば、$16 \times \frac{1}{4} = 4$ 倍の性能が得られる。第二に、GPU コアははるかに単純であり（実際、長い間、汎用コードを実行することすら できなかった）、そのためエネルギー効率が高いのである。たとえば、(i) 投機的実行を通常はサポートせず、(ii) 各処理要素を個別にプログラムすることは通常できず、(iii) コアごとのキャッシュははるかに小さい傾向がある。最後に、深層学習の多くの演算は高いメモリ帯域幅を必要とする。この点でも GPU は優れており、多くの CPU より少なくとも 10 倍広いバスを備えている。

2012 年に戻ろう。大きな突破口は、Alex Krizhevsky と Ilya Sutskever が GPU 上で動作する深い CNN を実装したときに訪れた。彼らは、CNN における計算上のボトルネックである畳み込みと行列乗算が、いずれもハードウェアで並列化できる演算であることに気づいた。3GB のメモリを持つ 2 枚の NVIDIA GTX 580 を用い、それぞれが 1.5 TFLOPs を実行可能であったため（10 年後でも多くの CPU にとっては依然として難題であった）、彼らは高速な畳み込みを実装した。cuda-convnet[77] のコードは十分に優れており、その後数年間にわたって業界標準となり、深層学習ブームの最初の数年間を支えた。

77 

11.1.2 AlexNet

8 層 CNN を採用した AlexNet は、ImageNet Large Scale Visual Recognition Challenge 2012 で大差をつけて優勝した (Russakovsky *et al.*, 2013)。このネットワークは、学習によって得られる特徴が手作業で設計された特徴を超えうることを、コンピュータビジョンにおいて初めて示し、従来のパラダイムを打ち破った。

AlexNet と LeNet のアーキテクチャは、Fig. 11.1.2 が示すように、驚くほど似ている。ここでは、2012 年当時にモデルを 2 つの小さな GPU に収めるために必要だったいくつかの設計上の癖を取り除いた、やや簡略化した AlexNet を示す。

AlexNet と LeNet の間には、重要な違いもある。第一に、AlexNet は比較的小さな LeNet-5 よりもはるかに深い。AlexNet は 8 層から成り、5 つの畳み込み層、2 つの全結合隠れ層、1 つの全結合出力層を含む。第二に、AlexNet は活性化関数として sigmoid ではなく ReLU を用いた。以下で詳細を見ていこう。

アーキテクチャ

AlexNet の第 1 層では、畳み込みウィンドウの形状は 11×11 である。ImageNet の画像は MNIST の画像よりも縦横ともに 8 倍大きいため、ImageNet データ中の

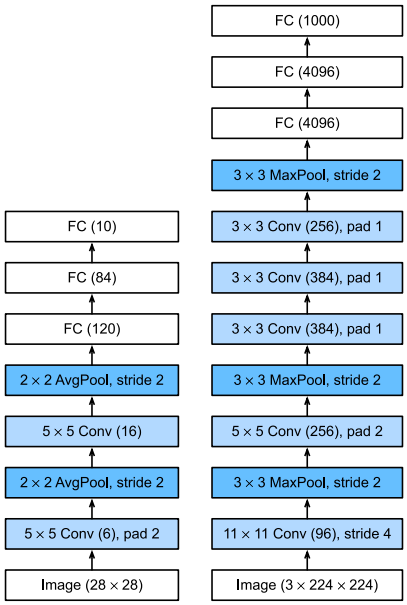


Fig. 11.1.2 LeNet（左）から AlexNet（右）へ。

物体は、より多くの画素とより豊かな視覚的詳細を占める傾向がある。したがって、物体を捉えるにはより大きな畳み込みウィンドウが必要である。第 2 層の畳み込みウィンドウの形状は 5×5 に縮小され、その後に 3×3 が続く。さらに、第 1、第 2、第 5 の畳み込み層の後には、ウィンドウ形状 3×3、ストライド 2 の最大プーリング層が追加される。また、AlexNet の畳み込みチャネル数は LeNet の 10 倍である。

最後の畳み込み層の後には、4096 出力を持つ非常に大きな全結合層が 2 つある。これらの層だけでほぼ 1GB のモデルパラメータを必要とする。初期の GPU はメモリが限られていたため、元の AlexNet ではデュアルデータストリーム設計が採用され、2 つの GPU がそれぞれモデルの半分だけを保存・計算する役割を担っていた。幸いなことに、現在は GPU メモリが比較的豊富なので、最近ではモデルを GPU 間で分割する必要はほとんどない（この点で、ここでの AlexNet モデルは元論文と異なる）。

活性化関数

さらに、AlexNet は sigmoid 活性化関数を、より単純な ReLU 活性化関数に変更した。第一に、ReLU 活性化関数の計算はより単純である。たとえば、sigmoid 活性化関数にある指数演算がない。第二に、ReLU 活性化関数は、異なるパラメータ初期化法を用いる場合にモデル訓練を容易にする。これは、sigmoid 活性化関数の出力が 0 または 1 に非常に近いとき、その領域の勾配はほぼ 0 となり、逆伝播がモデルパラメータの一部を更新し続けられなくなるためである。対照的に、ReLU 活性化関数の正の区間における勾配は常に 1 である（Section 8.1.2）。したがって、モデルパラメータが適切に初期化されていない場合、sigmoid 関数

では正の区間で勾配がほぼ 0 になり、モデルを効果的に訓練できない可能性がある。

容量制御と前処理

AlexNet は dropout（Section 8.6）によって全結合層のモデル複雑度を制御する一方、LeNet は重み減衰のみを使用する。さらにデータを増強するために、AlexNet の訓練ループでは、反転、切り抜き、色の変化など、多くの画像拡張が追加された。これによりモデルはより頑健になり、実質的なサンプルサイズの増加によって過学習が抑えられる。このような前処理手順の詳細なレビューについては Buslaev *et al.* (2020) を参照せよ。

```
class AlexNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nn.Conv2D(96, kernel_size=11, strides=4, activation='relu'),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Conv2D(256, kernel_size=5, padding=2, activation='relu'),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
                nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
                nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                nn.Dense(num_classes))
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nn.LazyConv2d(96, kernel_size=11, stride=4, padding=1),
                nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2),
                nn.LazyConv2d(256, kernel_size=5, padding=2), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2),
                nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
                nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
                nn.LazyConv2d(256, kernel_size=3, padding=1), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
                nn.LazyLinear(4096), nn.ReLU(),nn.Dropout(p=0.5),
                nn.LazyLinear(num_classes))
            self.net.apply(d2l.init_cnn)
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=96, kernel_size=11, strides=4,
                                       activation='relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                tf.keras.layers.Conv2D(filters=256, kernel_size=5, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
```

(continues on next page)

(continued from previous page)

```
                tf.keras.layers.Conv2D(filters=384, kernel_size=3, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.Conv2D(filters=384, kernel_size=3, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.Conv2D(filters=256, kernel_size=3, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                tf.keras.layers.Flatten(),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(num_classes)])
```

```
class AlexNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nn.Conv2D(96, kernel_size=11, strides=4, activation='relu'),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Conv2D(256, kernel_size=5, padding=2, activation='relu'),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
                nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
                nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                nn.Dense(num_classes))
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nn.LazyConv2d(96, kernel_size=11, stride=4, padding=1),
                nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2),
                nn.LazyConv2d(256, kernel_size=5, padding=2), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2),
                nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
                nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
                nn.LazyConv2d(256, kernel_size=3, padding=1), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
                nn.LazyLinear(4096), nn.ReLU(),nn.Dropout(p=0.5),
                nn.LazyLinear(num_classes))
            self.net.apply(d2l.init_cnn)
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=96, kernel_size=11, strides=4,
                                       activation='relu'),
```

(continues on next page)

(continued from previous page)

```
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                tf.keras.layers.Conv2D(filters=256, kernel_size=5, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                tf.keras.layers.Conv2D(filters=384, kernel_size=3, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.Conv2D(filters=384, kernel_size=3, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.Conv2D(filters=256, kernel_size=3, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                tf.keras.layers.Flatten(),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(num_classes)])
```

```
class AlexNet(d2l.Classifier):
    lr: float = 0.1
    num_classes: int = 10
    training: bool = True

    def setup(self):
        self.net = nn.Sequential([
            nn.Conv(features=96, kernel_size=(11, 11), strides=4, padding=1),
            nn.relu,
            lambda x: nn.max_pool(x, window_shape=(3, 3), strides=(2, 2)),
            nn.Conv(features=256, kernel_size=(5, 5)),
            nn.relu,
            lambda x: nn.max_pool(x, window_shape=(3, 3), strides=(2, 2)),
            nn.Conv(features=384, kernel_size=(3, 3)), nn.relu,
            nn.Conv(features=384, kernel_size=(3, 3)), nn.relu,
            nn.Conv(features=256, kernel_size=(3, 3)), nn.relu,
            lambda x: nn.max_pool(x, window_shape=(3, 3), strides=(2, 2)),
            lambda x: x.reshape((x.shape[0], -1)),  # flatten
            nn.Dense(features=4096),
            nn.relu,
            nn.Dropout(0.5, deterministic=not self.training),
            nn.Dense(features=4096),
            nn.relu,
            nn.Dropout(0.5, deterministic=not self.training),
            nn.Dense(features=self.num_classes)
        ])
```

```
class AlexNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
```

(continues on next page)

(continued from previous page)

```
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nn.Conv2D(96, kernel_size=11, strides=4, activation='relu'),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Conv2D(256, kernel_size=5, padding=2, activation='relu'),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
                nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
                nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                nn.Dense(num_classes))
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nn.LazyConv2d(96, kernel_size=11, stride=4, padding=1),
                nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2),
                nn.LazyConv2d(256, kernel_size=5, padding=2), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2),
                nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
                nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
                nn.LazyConv2d(256, kernel_size=3, padding=1), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
                nn.LazyLinear(4096), nn.ReLU(),nn.Dropout(p=0.5),
                nn.LazyLinear(num_classes))
            self.net.apply(d2l.init_cnn)
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=96, kernel_size=11, strides=4,
                                       activation='relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                tf.keras.layers.Conv2D(filters=256, kernel_size=5, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                tf.keras.layers.Conv2D(filters=384, kernel_size=3, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.Conv2D(filters=384, kernel_size=3, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.Conv2D(filters=256, kernel_size=3, padding=
↪'same',
                                       activation='relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                tf.keras.layers.Flatten(),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(num_classes)])
```

高さと幅がともに 224 の単一チャネルのデータ例を構成し各層の出力形状を観察するため、Fig. 11.1.2 の AlexNet アーキテクチャに対応させる。

```
AlexNet().layer_summary((1, 1, 224, 224))
```

```
Conv2d output shape:         torch.Size([1, 96, 54, 54])
ReLU output shape:   torch.Size([1, 96, 54, 54])
MaxPool2d output shape:      torch.Size([1, 96, 26, 26])
Conv2d output shape:         torch.Size([1, 256, 26, 26])
ReLU output shape:   torch.Size([1, 256, 26, 26])
MaxPool2d output shape:      torch.Size([1, 256, 12, 12])
Conv2d output shape:         torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
Conv2d output shape:         torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
Conv2d output shape:         torch.Size([1, 256, 12, 12])
ReLU output shape:   torch.Size([1, 256, 12, 12])
MaxPool2d output shape:      torch.Size([1, 256, 5, 5])
Flatten output shape:        torch.Size([1, 6400])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 10])
```

```
AlexNet().layer_summary((1, 1, 224, 224))
```

```
Conv2D output shape:         (1, 96, 54, 54)
MaxPool2D output shape:      (1, 96, 26, 26)
Conv2D output shape:         (1, 256, 26, 26)
MaxPool2D output shape:      (1, 256, 12, 12)
Conv2D output shape:         (1, 384, 12, 12)
Conv2D output shape:         (1, 384, 12, 12)
Conv2D output shape:         (1, 256, 12, 12)
MaxPool2D output shape:      (1, 256, 5, 5)
Dense output shape:  (1, 4096)
Dropout output shape:        (1, 4096)
Dense output shape:  (1, 4096)
Dropout output shape:        (1, 4096)
Dense output shape:  (1, 10)
[07:35:39] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
AlexNet(training=False).layer_summary((1, 224, 224, 1))
```

```
Conv output shape:   (1, 54, 54, 96)
custom_jvp output shape:     (1, 54, 54, 96)
function output shape:       (1, 26, 26, 96)
```

(continues on next page)

(continued from previous page)

```
Conv output shape:   (1, 26, 26, 256)
custom_jvp output shape:     (1, 26, 26, 256)
function output shape:       (1, 12, 12, 256)
Conv output shape:   (1, 12, 12, 384)
custom_jvp output shape:     (1, 12, 12, 384)
Conv output shape:   (1, 12, 12, 384)
custom_jvp output shape:     (1, 12, 12, 384)
Conv output shape:   (1, 12, 12, 256)
custom_jvp output shape:     (1, 12, 12, 256)
function output shape:       (1, 5, 5, 256)
function output shape:       (1, 6400)
Dense output shape:  (1, 4096)
custom_jvp output shape:     (1, 4096)
Dropout output shape:        (1, 4096)
Dense output shape:  (1, 4096)
custom_jvp output shape:     (1, 4096)
Dropout output shape:        (1, 4096)
Dense output shape:  (1, 10)
```

```
AlexNet().layer_summary((1, 224, 224, 1))
```

```
Conv2D output shape:         (1, 54, 54, 96)
MaxPooling2D output shape:   (1, 26, 26, 96)
Conv2D output shape:         (1, 26, 26, 256)
MaxPooling2D output shape:   (1, 12, 12, 256)
Conv2D output shape:         (1, 12, 12, 384)
Conv2D output shape:         (1, 12, 12, 384)
Conv2D output shape:         (1, 12, 12, 256)
MaxPooling2D output shape:   (1, 5, 5, 256)
Flatten output shape:        (1, 6400)
Dense output shape:  (1, 4096)
Dropout output shape:        (1, 4096)
Dense output shape:  (1, 4096)
Dropout output shape:        (1, 4096)
Dense output shape:  (1, 10)
```

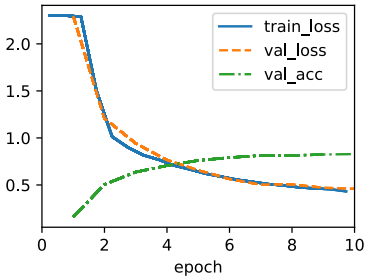
11.1.3 訓練

AlexNet は Krizhevsky *et al.* (2012) で ImageNet 上で訓練されたが、ここでは Fashion-MNIST を使う。ImageNet モデルを収束まで訓練するには、現代の GPU でも数時間から数日かかることがあるからである。AlexNet を Fashion-MNIST に直接適用する際の問題の一つは、その画像の解像度が低い（28×28 ピクセル）ImageNet の画像よりも。ことである。これを動作させるために、224×224 にアップサンプリングする。これは、情報を増やさずに計算複雑度を単に増やすだけなので、一般には賢明な方法ではない。それでも、AlexNet のアーキテクチャに忠実であるためにここではそうする。このリサイズは `d2l.FashionMNIST` コンストラクタの `resize` 引数で行う。

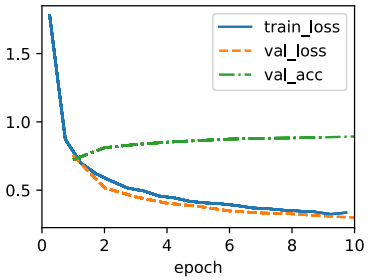
これで、AlexNet の訓練を開始できる。Section 10.6 の LeNet と比べると、ここでの主な変更点は、より小さな学習率の使用と、より深く広いネットワーク、よ

り高い画像解像度、そしてより高コストな畳み込みによる、はるかに遅い訓練である。

```
model = AlexNet(lr=0.01)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```



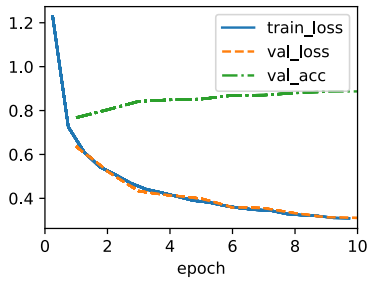

```
model = AlexNet(lr=0.01)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```

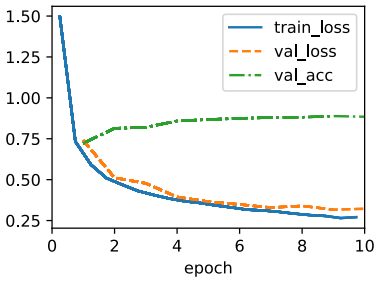



```
model = AlexNet(lr=0.01)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```

```
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
with d2l.try_gpu():
    model = AlexNet(lr=0.01)
    trainer.fit(model, data)
```

11.1.4 議論





AlexNet の構造は LeNet と驚くほど似ているが、精度向上（dropout）と訓練のしやすさ（ReLU）の両面でいくつかの重要な改良が加えられている。同様に驚くべきなのは、深層学習ツールの進歩の大きさである。2012 年には数か月かかった作業が、今ではどの現代的フレームワークでも十数行のコードで実現できる。

アーキテクチャを見直すと、AlexNet には効率面で弱点があることがわかる。最後の 2 つの隠れ層は、それぞれサイズ 6400×4096 と 4096×4096 の行列を必要とする。これは 164 MB のメモリと 81 MFLOPs の計算に相当し、どちらも、特に携帯電話のような小型デバイスでは無視できない負担である。これが、AlexNet が後続の節で扱う、はるかに効果的なアーキテクチャに取って代わられた理由の一つである。それでも、これは現在使われている浅いネットワークから深いネットワークへの重要な一歩である。実験では、パラメータ数が訓練データ量をはるかに上回っているにもかかわらず（最後の 2 層だけで 4000 万以上のパラメータがあり、6 万枚の画像からなるデータセットで訓練している）、過学習はほとんど見られない。訓練損失と検証損失は、訓練を通して事実上同一である。これは、現代の深層ネットワーク設計に本質的な dropout などの正則化が改善されたためである。

AlexNet の実装は LeNet より数行多いだけに見えるが、この概念的変化を受け入れ、その優れた実験結果を活用するまでに、学術界は長い年月を要した。これも、効率的な計算ツールが欠けていたためである。当時は DistBelief (Dean *et al.*, 2012) も Caffe (Jia *et al.*, 2014) も存在せず、Theano (Bergstra *et al.*, 2010) もまだ多くの特徴を欠いていた。状況を劇的に変えたのは、TensorFlow (Abadi *et al.*, 2016) の登場であった。

11.1.5 演習

1. 上の議論を踏まえて、AlexNet の計算特性を分析しなさい。
   1. 畳み込み層と全結合層それぞれのメモリ使用量を計算しなさい。どちらが支配的か。
   2. 畳み込み層と全結合層の計算コストを計算しなさい。
   3. メモリ（読み書き帯域幅、レイテンシ、サイズ）は計算にどのように影響するか。訓練と推論でその影響に違いはあるか。
2. あなたはチップ設計者であり、計算とメモリ帯域幅のトレードオフを考えなければならない。たとえば、より高速なチップはより多くの電力を消費し、場合によってはより大きなチップ面積を必要とする。より多いメモリ帯域幅は、より多くのピンと制御ロジックを必要とし、やはり面積を増やす。どのように最適化するか。
3. なぜエンジニアはもはや AlexNet の性能ベンチマークを報告しないのか。
4. AlexNet の訓練でエポック数を増やしてみなさい。LeNet と比べて、結果はどう異なるか。なぜか。
5. AlexNet は Fashion-MNIST データセットには複雑すぎるかもしれない。特に初期画像の解像度が低いためである。
   1. 訓練を速くしつつ、精度が大きく低下しないようにモデルを単純化してみなさい。
   2. 28×28 画像に直接適用できる、より良いモデルを設計しなさい。
6. バッチサイズを変更し、スループット（images/s）、精度、GPU メモリの変化を観察しなさい。
7. LeNet-5 に dropout と ReLU を適用しなさい。改善するか。画像に内在する不変性を利用するために前処理を行うことで、さらに改善できるか。
8. AlexNet を過学習させることはできるか。訓練を破綻させるには、どの特徴を取り除くか、あるいは変更する必要があるか。

11.2 ブロックを用いたネットワーク（VGG）

AlexNet は、深い CNN が良い結果を達成できるという実証的証拠を示したが、その後の研究者が新しいネットワークを設計する際の一般的なひな形は提示しませんでした。以下の節では、深層ネットワークの設計によく用いられるいくつかのヒューリスティックな概念を紹介する。

この分野の進歩は、チップ設計における VLSI（very large scale integration）の進

歩を思わせる。そこでは、エンジニアはトランジスタの配置から論理要素へ、さらに論理ブロックへと移っていった (Mead, 1980)。同様に、ニューラルネットワークアーキテクチャの設計も徐々に抽象化が進み、研究者は個々のニューロンから層全体へ、そして現在では層の反復パターンであるブロックへと考えるようになった。さらに 10 年後の現在では、研究者が学習済みのモデル全体を別の、とはいえ関連するタスクに転用するところまで進んでいる。このような大規模な事前学習済みモデルは、通常 *foundation models* と呼ばれる (Bommasani *et al.*, 2021)。

話をネットワーク設計に戻そう。ブロックを用いるという考え方は、オックスフォード大学の Visual Geometry Group（VGG）による、同名の *VGG* ネットワークで最初に現れた (Simonyan and Zisserman, 2014)。ループとサブルーチンを使えば、現代のどの深層学習フレームワークでも、こうした反復構造をコードで簡単に実装できる。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import np, npx, init
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
```

```
import tensorflow as tf
from d2l import tensorflow as d2l
```

11.2.1 VGG ブロック

CNN の基本的な構成要素は、次の順序からなる系列である。
(i) 解像度を維持するためにパディングを施した畳み込み層、
(ii) ReLU などの非線形性、
(iii) 解像度を下げるための max-pooling などのプーリング層。
この方法の問題の一つは、空間解像度がかなり急速に低下することである。特に、すべての次元（d）を使い切る前に、ネットワークに許される畳み込み層の数は $\log_2 d$ に厳しく制限される。たとえば ImageNet の場合、この方法では 8 層を超える畳み込み層を持つことは不可能である。

Simonyan and Zisserman (2014) の重要なアイデアは、max-pooling によるダウンサンプリングの間に、複数の畳み込みをブロックとして挟むことであった。彼ら

は主として、深いネットワークと幅広いネットワークのどちらがより良い性能を示すかに関心を持っていた。たとえば、2 つの 3×3 畳み込みを連続して適用すると、1 つの 5×5 畳み込みと同じ画素に触れる。同時に、後者はおよそ同じ数のパラメータ（$25 \cdot c^2$）を使うが、これは 3 つの 3×3 畳み込み（$3 \cdot 9 \cdot c^2$）とほぼ同じである。かなり詳細な解析により、深くて狭いネットワークが浅いネットワークよりも大幅に優れていることを示した。これにより深層学習は、典型的な応用で 100 層を超える、ますます深いネットワークを目指す方向へ進んだ。

3×3 畳み込みを積み重ねる設計は、その後の深層ネットワークにおける黄金標準となった（この設計判断が最近になって再検討されたのは Liu *et al.* (2022) である）。その結果、小さな畳み込みの高速実装は GPU の定番となった (Lavin and Gray, 2016)。

VGG に戻ると、VGG ブロックは、パディング 1 の 3×3 カーネルを持つ畳み込みの 系列に続いて、ストライド 2 の 2×2 max-pooling 層（各ブロック後に高さと幅を半分にする）から構成される。以下のコードでは、1 つの VGG ブロックを実装する `vgg_block` という関数を定義する。

以下の関数は 2 つの引数を取り、畳み込み層の数 `num_convs` と出力チャネル数 `num_channels` に対応する。

```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

```
def vgg_block(num_convs, num_channels):
    blk = nn.Sequential()
    for _ in range(num_convs):
        blk.add(nn.Conv2D(num_channels, kernel_size=3,
                          padding=1, activation='relu'))
    blk.add(nn.MaxPool2D(pool_size=2, strides=2))
    return blk
```

```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.Conv(out_channels, kernel_size=(3, 3), padding=(1,␣
↪1)))
        layers.append(nn.relu)
    layers.append(lambda x: nn.max_pool(x, window_shape=(2, 2), strides=(2,␣
↪2)))
    return nn.Sequential(layers)
```

```
def vgg_block(num_convs, num_channels):
    blk = tf.keras.models.Sequential()
    for _ in range(num_convs):
        blk.add(
            tf.keras.layers.Conv2D(num_channels, kernel_size=3,
                                   padding='same', activation='relu'))
    blk.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
    return blk
```

11.2.2 VGG ネットワーク

AlexNet や LeNet と同様に、VGG ネットワークは 2 つの部分に分けられる。
前半は主として畳み込み層とプーリング層からなり、後半は AlexNet と同じ全結合層からなる。
重要な違いは、畳み込み層が、次元を変えない非線形変換のまとまりとしてグループ化され、その後に解像度を下げるステップが続く点である。これは Fig. 11.2.1 に示されている。

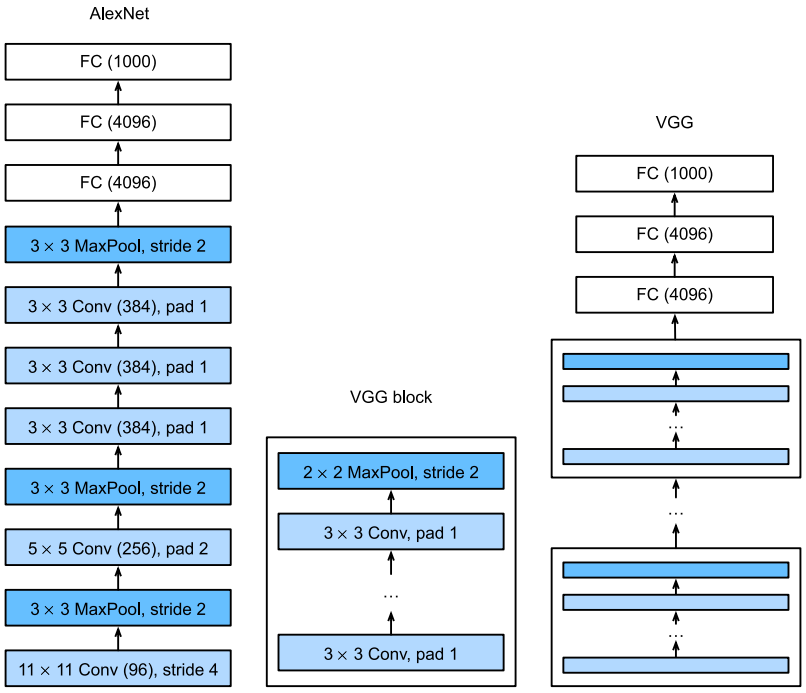


Fig. 11.2.1 AlexNet から VGG へ。重要な違いは、VGG が層のブロックから構成されるのに対し、AlexNet の層はすべて個別に設計されている点である。

ネットワークの畳み込み部分は、Fig. 11.2.1 にあるいくつかの VGG ブロック

（vgg_block 関数でも定義済み）を順番に接続したものである。この畳み込みのグループ化は、具体的な演算の選択は大きく変化してきたものの、過去 10 年ほとんど変わらずに残ってきたパターンである。

変数 arch はタプルのリスト（ブロックごとに 1 つ）で構成され、各タプルは 2 つの値、すなわち畳み込み層の数と出力チャネル数を含む。これはちょうど vgg_block 関数を呼び出すために必要な引数である。したがって、VGG は単一の具体的なモデルというより、ネットワークの ファミリーを定義している。特定のネットワークを構築するには、arch を順にたどってブロックを組み立てればよいだけである。

```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            for (num_convs, num_channels) in arch:
                self.net.add(vgg_block(num_convs, num_channels))
            self.net.add(nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                         nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                         nn.Dense(num_classes))
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
            conv_blks = []
            for (num_convs, out_channels) in arch:
                conv_blks.append(vgg_block(num_convs, out_channels))
            self.net = nn.Sequential(
                *conv_blks, nn.Flatten(),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(num_classes))
            self.net.apply(d2l.init_cnn)
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential()
            for (num_convs, num_channels) in arch:
                self.net.add(vgg_block(num_convs, num_channels))
            self.net.add(
                tf.keras.models.Sequential([
                tf.keras.layers.Flatten(),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(num_classes)]))
```

```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
```

(continues on next page)

(continued from previous page)

```
            for (num_convs, num_channels) in arch:
                self.net.add(vgg_block(num_convs, num_channels))
            self.net.add(nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                         nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                         nn.Dense(num_classes))
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
            conv_blks = []
            for (num_convs, out_channels) in arch:
                conv_blks.append(vgg_block(num_convs, out_channels))
            self.net = nn.Sequential(
                *conv_blks, nn.Flatten(),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(num_classes))
            self.net.apply(d2l.init_cnn)
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential()
            for (num_convs, num_channels) in arch:
                self.net.add(vgg_block(num_convs, num_channels))
            self.net.add(
                tf.keras.models.Sequential([
                tf.keras.layers.Flatten(),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(num_classes)]))
```

```
class VGG(d2l.Classifier):
    arch: list
    lr: float = 0.1
    num_classes: int = 10
    training: bool = True

    def setup(self):
        conv_blks = []
        for (num_convs, out_channels) in self.arch:
            conv_blks.append(vgg_block(num_convs, out_channels))

        self.net = nn.Sequential([
            *conv_blks,
            lambda x: x.reshape((x.shape[0], -1)),  # flatten
            nn.Dense(4096), nn.relu,
            nn.Dropout(0.5, deterministic=not self.training),
            nn.Dense(4096), nn.relu,
            nn.Dropout(0.5, deterministic=not self.training),
            nn.Dense(self.num_classes)])
```

```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
```

(continues on next page)

(continued from previous page)

```
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            for (num_convs, num_channels) in arch:
                self.net.add(vgg_block(num_convs, num_channels))
            self.net.add(nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                         nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                         nn.Dense(num_classes))
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
            conv_blks = []
            for (num_convs, out_channels) in arch:
                conv_blks.append(vgg_block(num_convs, out_channels))
            self.net = nn.Sequential(
                *conv_blks, nn.Flatten(),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(num_classes))
            self.net.apply(d2l.init_cnn)
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential()
            for (num_convs, num_channels) in arch:
                self.net.add(vgg_block(num_convs, num_channels))
            self.net.add(
                tf.keras.models.Sequential([
                tf.keras.layers.Flatten(),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(4096, activation='relu'),
                tf.keras.layers.Dropout(0.5),
                tf.keras.layers.Dense(num_classes)]))
```

元の VGG ネットワークは 5 つの畳み込みブロックからなり、そのうち最初の 2 つはそれぞれ 1 層の畳み込み層を持ち、残りの 3 つはそれぞれ 2 層の畳み込み層を含みる。最初のブロックの出力チャネル数は 64 で、その後の各ブロックでは出力チャネル数が 2 倍ずつ増え、512 に達するまで続く。このネットワークは 8 層の畳み込み層と 3 層の全結合層を使うため、しばしば VGG-11 と呼ばれる。

```
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 1, 224, 224))
```

```
Sequential output shape:     torch.Size([1, 64, 112, 112])
Sequential output shape:     torch.Size([1, 128, 56, 56])
Sequential output shape:     torch.Size([1, 256, 28, 28])
Sequential output shape:     torch.Size([1, 512, 14, 14])
Sequential output shape:     torch.Size([1, 512, 7, 7])
Flatten output shape:        torch.Size([1, 25088])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 4096])
```

(continues on next page)

(continued from previous page)

```
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 10])
```

```
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 1, 224, 224))
```

```
Sequential output shape:     (1, 64, 112, 112)
Sequential output shape:     (1, 128, 56, 56)
Sequential output shape:     (1, 256, 28, 28)
Sequential output shape:     (1, 512, 14, 14)
Sequential output shape:     (1, 512, 7, 7)
Dense output shape:  (1, 4096)
Dropout output shape:        (1, 4096)
Dense output shape:  (1, 4096)
Dropout output shape:        (1, 4096)
Dense output shape:  (1, 10)
[07:38:07] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512)),
    training=False).layer_summary((1, 224, 224, 1))
```

```
Sequential output shape:     (1, 112, 112, 64)
Sequential output shape:     (1, 56, 56, 128)
Sequential output shape:     (1, 28, 28, 256)
Sequential output shape:     (1, 14, 14, 512)
Sequential output shape:     (1, 7, 7, 512)
function output shape:       (1, 25088)
Dense output shape:  (1, 4096)
custom_jvp output shape:     (1, 4096)
Dropout output shape:        (1, 4096)
Dense output shape:  (1, 4096)
custom_jvp output shape:     (1, 4096)
Dropout output shape:        (1, 4096)
Dense output shape:  (1, 10)
```

```
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 224, 224, 1))
```

```
Sequential output shape:     (1, 112, 112, 64)
Sequential output shape:     (1, 56, 56, 128)
Sequential output shape:     (1, 28, 28, 256)
Sequential output shape:     (1, 14, 14, 512)
Sequential output shape:     (1, 7, 7, 512)
Sequential output shape:     (1, 10)
```

見てわかるように、各ブロックで高さと幅を半分にし、最終的に高さと幅が 7 になってから表現を平坦化し、ネットワークの全結合部分で処理する。

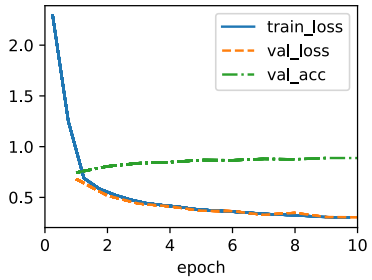
Simonyan and Zisserman (2014) では、VGG の他のいくつかの変種も説明されている。実際、新しいアーキテクチャを導入する際には、速度と精度のトレードオフが異なる ファミリーのネットワークを提案するのが今では標準になっている。

11.2.3 学習

VGG-11 は AlexNet より計算負荷が高いため、より少ないチャネル数のネットワークを構成する。これは Fashion-MNIST の学習には十分すぎるほどである。モデル学習の過程は Section 11.1 における AlexNet と同様である。

ここでも、検証損失と訓練損失がよく一致しており、過学習はわずかであることが示唆される。

```
model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```

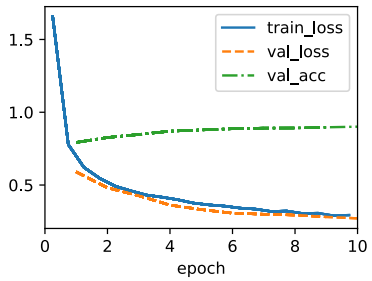



```
model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



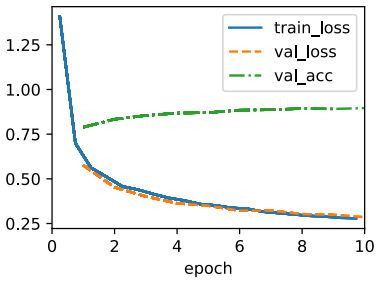
```
model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
if tab.selected('pytorch'):
```


(continues on next page)

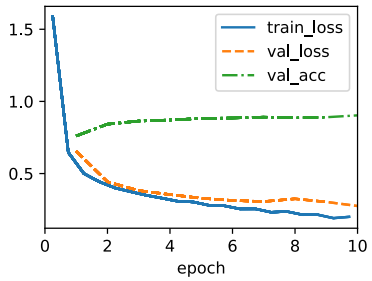


(continued from previous page)

```
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```




```
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
with d2l.try_gpu():
    model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.01)
    trainer.fit(model, data)
```



11.2.4 要約

VGG は、真に現代的な畳み込みニューラルネットワークの最初のものだと主張できるかもしれない。AlexNet は、深層学習を大規模で有効にする多くの構成要素を導入したが、複数の畳み込みからなるブロックや、深くて狭いネットワー

クを好む設計など、重要な性質を導入したのは VGG だと言えるだろう。また、実際に同じパラメータ化を持つモデルのファミリー全体である最初のネットワークでもあり、実践者に対して複雑さと速度の間で十分なトレードオフを与える。ここは、現代の深層学習フレームワークが真価を発揮する場でもある。もはやネットワークを指定するために XML の設定ファイルを生成する必要はなく、単純な Python コードでそのようなネットワークを組み立てればよいのである。

より最近では、ParNet (Goyal *et al.*, 2021) が、多数の並列計算を用いることで、はるかに浅いアーキテクチャでも競争力のある性能を達成できることを示した。これは刺激的な進展であり、将来のアーキテクチャ設計に影響を与えることが期待される。とはいえ、この章の残りでは、過去 10 年の科学的進歩の道筋に従うことにする。

11.2.5 演習

1. AlexNet と比べると、VGG は計算の面でかなり遅く、GPU メモリもより多く必要とする。
   1. AlexNet と VGG に必要なパラメータ数を比較せよ。
   2. 畳み込み層と全結合層で使われる浮動小数点演算回数を比較せよ。
   3. 全結合層によって生じる計算コストをどのように削減できるか。
2. ネットワークの各層に対応する次元を表示すると、ネットワークは 11 層あるにもかかわらず、8 つのブロック（に加えていくつかの補助変換）に関する情報しか見えない。残りの 3 層はどこへ行ったのだろうか。
3. VGG 論文 (Simonyan and Zisserman, 2014) の表 1 を用いて、VGG-16 や VGG-19 などの他の一般的なモデルを構成せよ。
4. Fashion-MNIST の解像度を 28×28 から 224×224 へ 8 倍にアップサンプリングするのは非常に無駄である。ネットワークアーキテクチャと解像度変換を変更して、たとえば入力を 56 または 84 次元にしてみよ。ネットワークの精度を下げずにそれを実現できるか。ダウンサンプリングの前により多くの非線形性を加える方法については、VGG 論文 (Simonyan and Zisserman, 2014) を参照せよ。

11.3 Network in Network（NiN）

LeNet、AlexNet、VGG はいずれも共通の設計パターンを共有している。すなわち、畳み込み層とプーリング層の系列を通じて 空間的構造を利用して特徴を抽出し、その後、全結合層によって表現を後処理する。AlexNet と VGG による LeNet からの改良は主として、これら後続のネットワークがこの 2 つのモジュールをどのように広げ、深くしたかにある。

この設計には 2 つの大きな課題がある。第 1 に、アーキテクチャの最後にある全結合層は、膨大な数のパラメータを消費する。たとえば、VGG-11 のような単純なモデルでさえ、単精度（FP32）では約 400MB の RAM を占有する巨大な行列を必要とする。これは、特にモバイル機器や組み込み機器では、計算に対する大きな障害となる。そもそも、高性能なスマートフォンであっても RAM は 8GB を超えない。VGG が考案された当時はそれより 1 桁少なく（iPhone 4S は 512MB だった）、画像分類器にメモリの大半を費やすことを正当化するのは難しかっただろう。

第 2 に、非線形性の程度を高めるために、ネットワークのより前段に全結合層を追加することも同様に不可能である。そうすると空間構造が失われ、さらに多くのメモリを要する可能性があるからだ。

network in network（*NiN*）ブロック (Lin *et al.*, 2013) は、これら 2 つの問題を 1 つの単純な戦略で解決できる代替案を提供する。これは非常に単純な洞察に基づいて提案された。(i) 1×1 畳み込みを用いてチャネル活性化全体に局所的な非線形性を追加し、(ii) 最後の表現層のすべての位置を統合するためにグローバル平均プーリングを用いる。なお、追加された非線形性がなければ、グローバル平均プーリングは有効ではない。以下でこれを詳しく見ていこう。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import np, npx, init
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
import tensorflow as tf
from d2l import tensorflow as d2l
```

11.3.1 NiN ブロック

Section 10.4.3 を思い出そう。そこで、畳み込み層の入力と出力は、例、チャネル、高さ、幅に対応する軸を持つ 4 次元テンソルからなると述べた。また、全結合層の入力と出力は、通常、例と特徴に対応する 2 次元テンソルであることも思い出してほしい。NiN の考え方は、各ピクセル位置（高さと幅の各位置）に対して全結合層を適用することである。その結果として得られる 1×1 畳み込みは、各ピクセル位置で独立に作用する全結合層とみなせる。

Fig. 11.3.1 は、VGG と NiN の主な構造上の違いと、それぞれのブロックを示し

ている。NiN ブロックの違い（最初の畳み込みの後に 1×1 畳み込みが続くのに対し、VGG では 3×3 畳み込みを維持する）と、最後に巨大な全結合層が不要になっている点の両方に注目してほしい。

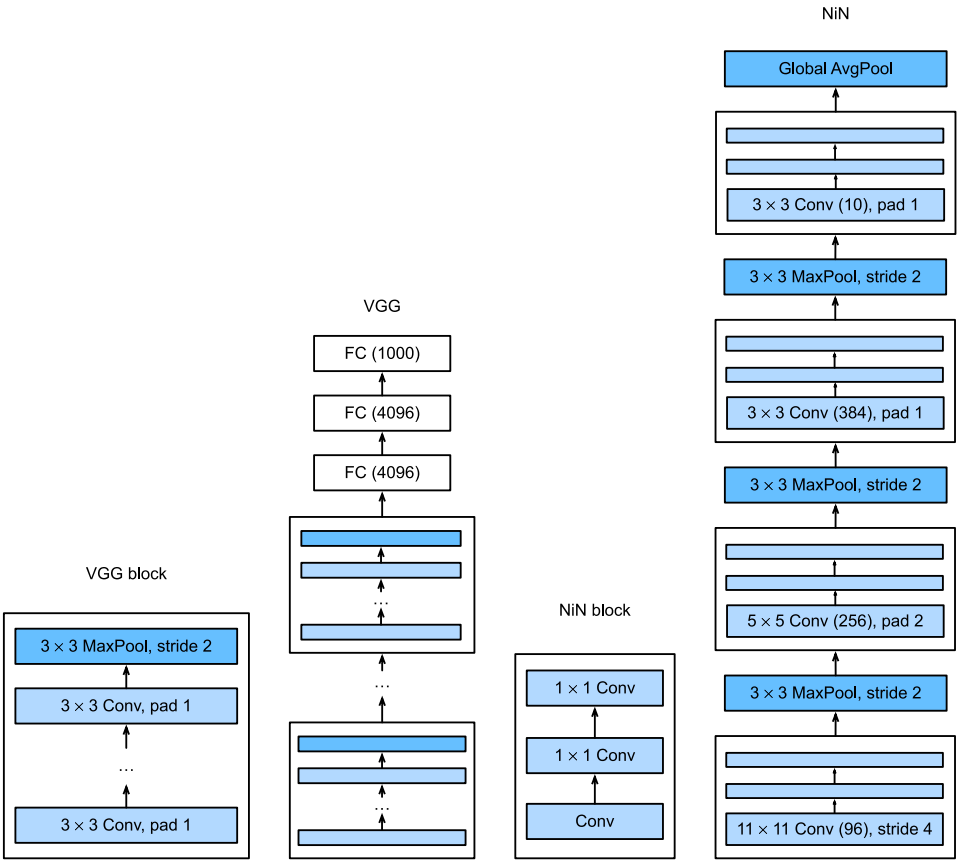


Fig. 11.3.1 VGG と NiN のアーキテクチャ、およびそれぞれのブロックの比較。

```
def nin_block(out_channels, kernel_size, strides, padding):
    return nn.Sequential(
        nn.LazyConv2d(out_channels, kernel_size, strides, padding), nn.ReLU(),
        nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU(),
        nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU())
```

```
def nin_block(num_channels, kernel_size, strides, padding):
    blk = nn.Sequential()
    blk.add(nn.Conv2D(num_channels, kernel_size, strides, padding,
                      activation='relu'),
            nn.Conv2D(num_channels, kernel_size=1, activation='relu'),
            nn.Conv2D(num_channels, kernel_size=1, activation='relu'))
    return blk
```

```
def nin_block(out_channels, kernel_size, strides, padding):
    return nn.Sequential([
        nn.Conv(out_channels, kernel_size, strides, padding),
        nn.relu,
        nn.Conv(out_channels, kernel_size=(1, 1)), nn.relu,
        nn.Conv(out_channels, kernel_size=(1, 1)), nn.relu])
```

```
def nin_block(out_channels, kernel_size, strides, padding):
    return tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(out_channels, kernel_size, strides=strides,
                           padding=padding),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.Conv2D(out_channels, 1),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.Conv2D(out_channels, 1),
    tf.keras.layers.Activation('relu')])
```

11.3.2 NiN モデル

NiN は AlexNet と同じ初期の畳み込みサイズを用いる（NiN はその直後に提案された）。カーネルサイズはそれぞれ 11×11、5×5、3×3 であり、出力チャネル数は AlexNet と一致する。各 NiN ブロックの後には、ストライド 2、ウィンドウ形状 3×3 の最大プーリング層が続く。

NiN と AlexNet および VGG の第 2 の重要な違いは、NiN が全結合層を完全に避けることである。その代わりに、NiN はラベルクラス数と等しい出力チャネル数を持つ NiN ブロックの後に、グローバル平均プーリング層を置き、ロジットのベクトルを得る。この設計は、学習時間が増える可能性という代償はあるものの、必要なモデルパラメータ数を大幅に削減する。

```
class NiN(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nin_block(96, kernel_size=11, strides=4, padding=0),
                nn.MaxPool2D(pool_size=3, strides=2),
                nin_block(256, kernel_size=5, strides=1, padding=2),
                nn.MaxPool2D(pool_size=3, strides=2),
                nin_block(384, kernel_size=3, strides=1, padding=1),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Dropout(0.5),
                nin_block(num_classes, kernel_size=3, strides=1, padding=1),
                nn.GlobalAvgPool2D(),
                nn.Flatten())
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nin_block(96, kernel_size=11, strides=4, padding=0),
```

(continues on next page)

(continued from previous page)

```
                nn.MaxPool2d(3, stride=2),
                nin_block(256, kernel_size=5, strides=1, padding=2),
                nn.MaxPool2d(3, stride=2),
                nin_block(384, kernel_size=3, strides=1, padding=1),
                nn.MaxPool2d(3, stride=2),
                nn.Dropout(0.5),
                nin_block(num_classes, kernel_size=3, strides=1, padding=1),
                nn.AdaptiveAvgPool2d((1, 1)),
                nn.Flatten())
            self.net.apply(d2l.init_cnn)
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                nin_block(96, kernel_size=11, strides=4, padding='valid'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                nin_block(256, kernel_size=5, strides=1, padding='same'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                nin_block(384, kernel_size=3, strides=1, padding='same'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                tf.keras.layers.Dropout(0.5),
                nin_block(num_classes, kernel_size=3, strides=1, padding='same
↪'),
                tf.keras.layers.GlobalAvgPool2D(),
                tf.keras.layers.Flatten()])
```

```
class NiN(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nin_block(96, kernel_size=11, strides=4, padding=0),
                nn.MaxPool2D(pool_size=3, strides=2),
                nin_block(256, kernel_size=5, strides=1, padding=2),
                nn.MaxPool2D(pool_size=3, strides=2),
                nin_block(384, kernel_size=3, strides=1, padding=1),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Dropout(0.5),
                nin_block(num_classes, kernel_size=3, strides=1, padding=1),
                nn.GlobalAvgPool2D(),
                nn.Flatten())
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nin_block(96, kernel_size=11, strides=4, padding=0),
                nn.MaxPool2d(3, stride=2),
                nin_block(256, kernel_size=5, strides=1, padding=2),
                nn.MaxPool2d(3, stride=2),
                nin_block(384, kernel_size=3, strides=1, padding=1),
                nn.MaxPool2d(3, stride=2),
                nn.Dropout(0.5),
                nin_block(num_classes, kernel_size=3, strides=1, padding=1),
                nn.AdaptiveAvgPool2d((1, 1)),
                nn.Flatten())
```

(continues on next page)

(continued from previous page)

```
            self.net.apply(d2l.init_cnn)
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                nin_block(96, kernel_size=11, strides=4, padding='valid'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                nin_block(256, kernel_size=5, strides=1, padding='same'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                nin_block(384, kernel_size=3, strides=1, padding='same'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                tf.keras.layers.Dropout(0.5),
                nin_block(num_classes, kernel_size=3, strides=1, padding='same
↪'),
                tf.keras.layers.GlobalAvgPool2D(),
                tf.keras.layers.Flatten()])
```

```
class NiN(d2l.Classifier):
    lr: float = 0.1
    num_classes = 10
    training: bool = True

    def setup(self):
        self.net = nn.Sequential([
            nin_block(96, kernel_size=(11, 11), strides=(4, 4), padding=(0,␣
↪0)),
            lambda x: nn.max_pool(x, (3, 3), strides=(2, 2)),
            nin_block(256, kernel_size=(5, 5), strides=(1, 1), padding=(2, 2)),
            lambda x: nn.max_pool(x, (3, 3), strides=(2, 2)),
            nin_block(384, kernel_size=(3, 3), strides=(1, 1), padding=(1, 1)),
            lambda x: nn.max_pool(x, (3, 3), strides=(2, 2)),
            nn.Dropout(0.5, deterministic=not self.training),
            nin_block(self.num_classes, kernel_size=(3, 3), strides=1,␣
↪padding=(1, 1)),
            lambda x: nn.avg_pool(x, (5, 5)),  # global avg pooling
            lambda x: x.reshape((x.shape[0], -1))  # flatten
        ])
```

```
class NiN(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nin_block(96, kernel_size=11, strides=4, padding=0),
                nn.MaxPool2D(pool_size=3, strides=2),
                nin_block(256, kernel_size=5, strides=1, padding=2),
                nn.MaxPool2D(pool_size=3, strides=2),
                nin_block(384, kernel_size=3, strides=1, padding=1),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Dropout(0.5),
                nin_block(num_classes, kernel_size=3, strides=1, padding=1),
                nn.GlobalAvgPool2D(),
                nn.Flatten())
```

(continues on next page)

(continued from previous page)

```
            self.net.initialize(init.Xavier())
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nin_block(96, kernel_size=11, strides=4, padding=0),
                nn.MaxPool2d(3, stride=2),
                nin_block(256, kernel_size=5, strides=1, padding=2),
                nn.MaxPool2d(3, stride=2),
                nin_block(384, kernel_size=3, strides=1, padding=1),
                nn.MaxPool2d(3, stride=2),
                nn.Dropout(0.5),
                nin_block(num_classes, kernel_size=3, strides=1, padding=1),
                nn.AdaptiveAvgPool2d((1, 1)),
                nn.Flatten())
            self.net.apply(d2l.init_cnn)
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                nin_block(96, kernel_size=11, strides=4, padding='valid'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                nin_block(256, kernel_size=5, strides=1, padding='same'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                nin_block(384, kernel_size=3, strides=1, padding='same'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2),
                tf.keras.layers.Dropout(0.5),
                nin_block(num_classes, kernel_size=3, strides=1, padding='same
↪'),
                tf.keras.layers.GlobalAvgPool2D(),
                tf.keras.layers.Flatten()])
```

各ブロックの出力形状を確認するためにデータ例を作成する。

```
NiN().layer_summary((1, 1, 224, 224))
```

```
Sequential output shape:     torch.Size([1, 96, 54, 54])
MaxPool2d output shape:      torch.Size([1, 96, 26, 26])
Sequential output shape:     torch.Size([1, 256, 26, 26])
MaxPool2d output shape:      torch.Size([1, 256, 12, 12])
Sequential output shape:     torch.Size([1, 384, 12, 12])
MaxPool2d output shape:      torch.Size([1, 384, 5, 5])
Dropout output shape:        torch.Size([1, 384, 5, 5])
Sequential output shape:     torch.Size([1, 10, 5, 5])
AdaptiveAvgPool2d output shape:      torch.Size([1, 10, 1, 1])
Flatten output shape:        torch.Size([1, 10])
```

```
NiN().layer_summary((1, 1, 224, 224))
```

```
Sequential output shape:     (1, 96, 54, 54)
MaxPool2D output shape:      (1, 96, 26, 26)
Sequential output shape:     (1, 256, 26, 26)
MaxPool2D output shape:      (1, 256, 12, 12)
Sequential output shape:     (1, 384, 12, 12)
```

(continues on next page)

(continued from previous page)

```
MaxPool2D output shape:      (1, 384, 5, 5)
Dropout output shape:        (1, 384, 5, 5)
Sequential output shape:     (1, 10, 5, 5)
GlobalAvgPool2D output shape:        (1, 10, 1, 1)
Flatten output shape:        (1, 10)
[07:48:54] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
NiN(training=False).layer_summary((1, 224, 224, 1))
```

```
Sequential output shape:     (1, 54, 54, 96)
function output shape:       (1, 26, 26, 96)
Sequential output shape:     (1, 26, 26, 256)
function output shape:       (1, 12, 12, 256)
Sequential output shape:     (1, 12, 12, 384)
function output shape:       (1, 5, 5, 384)
Dropout output shape:        (1, 5, 5, 384)
Sequential output shape:     (1, 5, 5, 10)
function output shape:       (1, 1, 1, 10)
function output shape:       (1, 10)
```

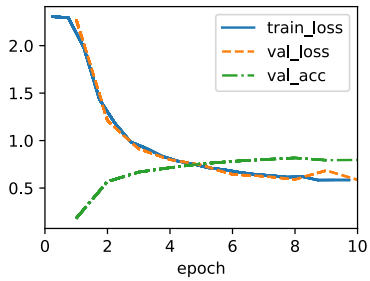
```
NiN().layer_summary((1, 224, 224, 1))
```

```
Sequential output shape:     (1, 54, 54, 96)
MaxPooling2D output shape:   (1, 26, 26, 96)
Sequential output shape:     (1, 26, 26, 256)
MaxPooling2D output shape:   (1, 12, 12, 256)
Sequential output shape:     (1, 12, 12, 384)
MaxPooling2D output shape:   (1, 5, 5, 384)
Dropout output shape:        (1, 5, 5, 384)
Sequential output shape:     (1, 5, 5, 10)
GlobalAveragePooling2D output shape:         (1, 10)
Flatten output shape:        (1, 10)
```

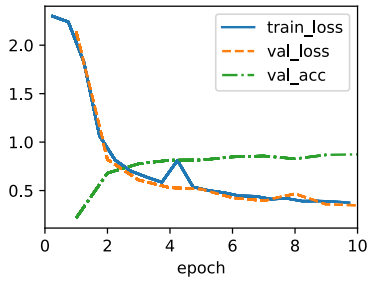
11.3.3 学習

これまでと同様に、AlexNet と VGG で用いたのと同じオプティマイザを使って Fashion-MNIST でモデルを学習する。

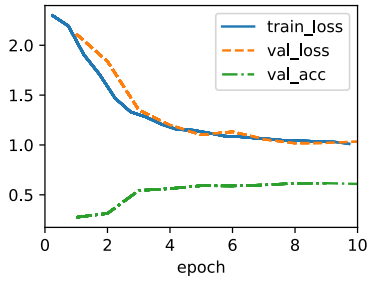
```
model = NiN(lr=0.05)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



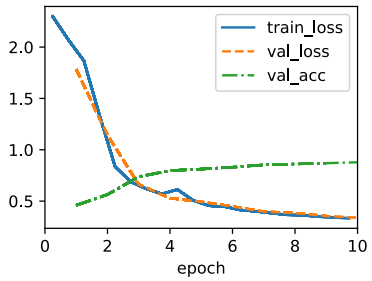

```
model = NiN(lr=0.05)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```




```
model = NiN(lr=0.05)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



```
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
with d2l.try_gpu():
    model = NiN(lr=0.05)
    trainer.fit(model, data)
```



11.3.4 まとめ

NiN は AlexNet や VGG よりも劇的に少ないパラメータしか持たない。これは主として、巨大な全結合層を必要としないことに由来する。その代わりに、ネットワーク本体の最後の段の後で、すべての画像位置を集約するためにグローバル平均プーリングを用いる。これにより、高価な（学習された）縮約演算が不要になり、単純な平均に置き換えられる。当時研究者を驚かせたのは、この平均化操作が精度を損なわなかったことである。低解像度の表現（多くのチャネルを持つ）に対して平均を取ることは、ネットワークが扱える平行移動不変性の量を増やすことにも注意してほしい。

広いカーネルを持つ畳み込みを減らし、それらを 1×1 畳み込みに置き換えることは、さらにパラメータ削減に役立つ。これは、任意の位置におけるチャネル間でかなりの非線形性を持たせることができる。1×1 畳み込みとグローバル平均プーリングの両方が、その後の CNN 設計に大きな影響を与えた。

11.3.5 演習

1. NiN ブロックに 1×1 畳み込み層が 2 つあるのはなぜか。3 つに増やすとどうなるか。1 つに減らすとどうなるか。何が変わるか。
2. 1×1 畳み込みを 3×3 畳み込みに置き換えると何が変わるか。
3. グローバル平均プーリングを全結合層に置き換えるとどうなるか（速度、精度、パラメータ数）。
4. NiN のリソース使用量を計算せよ。
    1. パラメータ数はいくつか。
    2. 計算量はどれくらいか。
    3. 学習中に必要なメモリ量はどれくらいか。

4. 予測時に必要なメモリ量はどれくらいか。
5. $384 \times 5 \times 5$ の表現を一度に $10 \times 5 \times 5$ の表現へ縮約することには、どのような問題がありうるか。
6. VGG-11、VGG-16、VGG-19 を生み出した VGG の構造設計上の決定を用いて、NiN 風ネットワークの系列を設計せよ。

11.4 マルチブランチネットワーク（GoogLeNet）

2014 年、*GoogLeNet* は ImageNet Challenge (Szegedy *et al.*, 2015) で優勝し、NiN (Lin *et al.*, 2013)、反復ブロック (Simonyan and Zisserman, 2014)、そしてさまざまな畳み込みカーネルを組み合わせた構造を用いた。おそらくこれは、CNN において stem（データ取り込み）、body（データ処理）、head（予測）を明確に区別した最初のネットワークでもあった。この設計パターンは、それ以来ディープネットワークの設計において定着している。*stem* は画像に対して作用する最初の 2 つまたは 3 つの畳み込み層で構成され、元の画像から低レベル特徴を抽出する。これに続くのが畳み込みブロックからなる *body* である。最後に *head* が、これまでに得られた特徴を、現在扱っている分類・セグメンテーション・検出・追跡の問題に応じた出力へ写像する。

GoogLeNet における重要な貢献は、ネットワーク本体の設計であった。これは畳み込みカーネルの選択という問題を独創的な方法で解決した。1×1 から 11×11 までのどの畳み込みが最適かを特定しようとした他の研究とは異なり、GoogLeNet は単にマルチブランチ畳み込みを 連結した。以下では、GoogLeNet のやや簡略化した版を紹介する。元の設計には、ネットワークの複数層に適用される中間損失関数を通じて学習を安定化させるためのいくつかの工夫が含まれていた。より優れた学習アルゴリズムが利用可能になった現在では、それらはもはや必要ない。

```
from d2l import torch as d2l
import torch
from torch import nn
from torch.nn import functional as F
```

```
from d2l import mxnet as d2l
from mxnet import np, npx, init
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from jax import numpy as jnp
import jax
```

```
import tensorflow as tf
from d2l import tensorflow as d2l
```

11.4.1 Inception ブロック

GoogLeNet の基本的な畳み込みブロックは *Inception block* と呼ばれ、映画 *Inception* のミーム「we need to go deeper」に由来する。

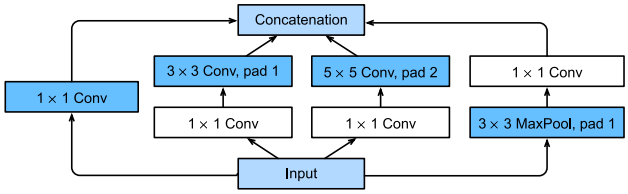


Fig. 11.4.1 Inception ブロックの構造。

Fig. 11.4.1 に示すように、Inception ブロックは 4 つの並列ブランチから構成される。最初の 3 つのブランチは、それぞれウィンドウサイズが 1×1、3×3、5×5 の畳み込み層を用いて、異なる空間スケールの情報を抽出する。中央の 2 つのブランチでは、入力に対して 1×1 畳み込みも追加してチャネル数を削減し、モデルの複雑さを抑えている。4 つ目のブランチは 3×3 の最大プーリング層を使い、その後に 1×1 畳み込み層を続けてチャネル数を変更する。4 つのブランチはいずれも適切なパディングを用いて、入力と出力の高さ・幅が同じになるようにする。最後に、各ブランチの出力をチャネル次元に沿って連結し、ブロックの出力とする。Inception ブロックで一般に調整されるハイパーパラメータは、各層の出力チャネル数、すなわち異なるサイズの畳み込みにどのように容量を割り当てるかである。

```
class Inception(nn.Module):
    # c1--c4 are the number of output channels for each branch
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Branch 1
        self.b1_1 = nn.LazyConv2d(c1, kernel_size=1)
        # Branch 2
        self.b2_1 = nn.LazyConv2d(c2[0], kernel_size=1)
        self.b2_2 = nn.LazyConv2d(c2[1], kernel_size=3, padding=1)
        # Branch 3
        self.b3_1 = nn.LazyConv2d(c3[0], kernel_size=1)
        self.b3_2 = nn.LazyConv2d(c3[1], kernel_size=5, padding=2)
        # Branch 4
        self.b4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.b4_2 = nn.LazyConv2d(c4, kernel_size=1)

    def forward(self, x):
        b1 = F.relu(self.b1_1(x))
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))
```

(continues on next page)

(continued from previous page)

```
        b4 = F.relu(self.b4_2(self.b4_1(x)))
        return torch.cat((b1, b2, b3, b4), dim=1)
```

```
class Inception(nn.Block):
    # c1--c4 are the number of output channels for each branch
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Branch 1
        self.b1_1 = nn.Conv2D(c1, kernel_size=1, activation='relu')
        # Branch 2
        self.b2_1 = nn.Conv2D(c2[0], kernel_size=1, activation='relu')
        self.b2_2 = nn.Conv2D(c2[1], kernel_size=3, padding=1,
                              activation='relu')
        # Branch 3
        self.b3_1 = nn.Conv2D(c3[0], kernel_size=1, activation='relu')
        self.b3_2 = nn.Conv2D(c3[1], kernel_size=5, padding=2,
                              activation='relu')
        # Branch 4
        self.b4_1 = nn.MaxPool2D(pool_size=3, strides=1, padding=1)
        self.b4_2 = nn.Conv2D(c4, kernel_size=1, activation='relu')

    def forward(self, x):
        b1 = self.b1_1(x)
        b2 = self.b2_2(self.b2_1(x))
        b3 = self.b3_2(self.b3_1(x))
        b4 = self.b4_2(self.b4_1(x))
        return np.concatenate((b1, b2, b3, b4), axis=1)
```

```
class Inception(nn.Module):
    # `c1`--`c4` are the number of output channels for each branch
    c1: int
    c2: tuple
    c3: tuple
    c4: int

    def setup(self):
        # Branch 1
        self.b1_1 = nn.Conv(self.c1, kernel_size=(1, 1))
        # Branch 2
        self.b2_1 = nn.Conv(self.c2[0], kernel_size=(1, 1))
        self.b2_2 = nn.Conv(self.c2[1], kernel_size=(3, 3), padding='same')
        # Branch 3
        self.b3_1 = nn.Conv(self.c3[0], kernel_size=(1, 1))
        self.b3_2 = nn.Conv(self.c3[1], kernel_size=(5, 5), padding='same')
        # Branch 4
        self.b4_1 = lambda x: nn.max_pool(x, window_shape=(3, 3),
                                          strides=(1, 1), padding='same')
        self.b4_2 = nn.Conv(self.c4, kernel_size=(1, 1))

    def __call__(self, x):
        b1 = nn.relu(self.b1_1(x))
        b2 = nn.relu(self.b2_2(nn.relu(self.b2_1(x))))
        b3 = nn.relu(self.b3_2(nn.relu(self.b3_1(x))))
```

(continues on next page)

(continued from previous page)

```
        b4 = nn.relu(self.b4_2(self.b4_1(x)))
        return jnp.concatenate((b1, b2, b3, b4), axis=-1)
```

```
class Inception(tf.keras.Model):
    # c1--c4 are the number of output channels for each branch
    def __init__(self, c1, c2, c3, c4):
        super().__init__()
        self.b1_1 = tf.keras.layers.Conv2D(c1, 1, activation='relu')
        self.b2_1 = tf.keras.layers.Conv2D(c2[0], 1, activation='relu')
        self.b2_2 = tf.keras.layers.Conv2D(c2[1], 3, padding='same',
                                           activation='relu')
        self.b3_1 = tf.keras.layers.Conv2D(c3[0], 1, activation='relu')
        self.b3_2 = tf.keras.layers.Conv2D(c3[1], 5, padding='same',
                                           activation='relu')
        self.b4_1 = tf.keras.layers.MaxPool2D(3, 1, padding='same')
        self.b4_2 = tf.keras.layers.Conv2D(c4, 1, activation='relu')

    def call(self, x):
        b1 = self.b1_1(x)
        b2 = self.b2_2(self.b2_1(x))
        b3 = self.b3_2(self.b3_1(x))
        b4 = self.b4_2(self.b4_1(x))
        return tf.keras.layers.Concatenate()([b1, b2, b3, b4])
```

このネットワークがなぜこれほどよく機能するのかを直感的に理解するために、フィルタの組み合わせを考えてみよう。さまざまなサイズのフィルタで画像を探索しているのである。つまり、異なる広がりの詳細を、それぞれに適したサイズのフィルタで効率よく認識できる。同時に、異なるフィルタに対して異なる量のパラメータを割り当てることもできる。

11.4.2 GoogLeNet モデル

Fig. 11.4.2 に示すように、GoogLeNet は合計 9 個の inception block を積み重ねたもので、3 つのグループに分かれ、その間に max-pooling を挟み、head では global average pooling を用いて推定値を生成する。Inception ブロック間の max-pooling は次元を削減する。stem では、最初のモジュールは AlexNet や LeNet に似ている。

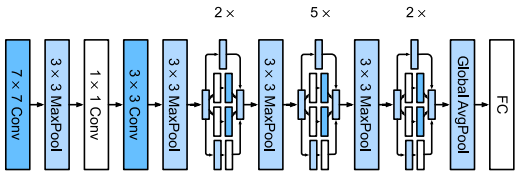


Fig. 11.4.2 GoogLeNet のアーキテクチャ。

ここから GoogLeNet を少しずつ実装していきよう。まず stem から始める。最初のモジュールは 64 チャネルの 7×7 畳み込み層を使う。

```
class GoogleNet(d2l.Classifier):
    def b1(self):
        if tab.selected('mxnet'):
            net = nn.Sequential()
            net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3,
                              activation='relu'),
                    nn.MaxPool2D(pool_size=3, strides=2, padding=1))
            return net
        if tab.selected('pytorch'):
            return nn.Sequential(
                nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        if tab.selected('tensorflow'):
            return tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(64, 7, strides=2, padding='same',
                                       activation='relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2,
                                          padding='same')])
```

```
class GoogleNet(d2l.Classifier):
    def b1(self):
        if tab.selected('mxnet'):
            net = nn.Sequential()
            net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3,
                              activation='relu'),
                    nn.MaxPool2D(pool_size=3, strides=2, padding=1))
            return net
        if tab.selected('pytorch'):
            return nn.Sequential(
                nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        if tab.selected('tensorflow'):
            return tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(64, 7, strides=2, padding='same',
                                       activation='relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2,
                                          padding='same')])
```

```
class GoogleNet(d2l.Classifier):
    lr: float = 0.1
    num_classes: int = 10

    def setup(self):
        self.net = nn.Sequential([self.b1(), self.b2(), self.b3(), self.b4(),
                                  self.b5(), nn.Dense(self.num_classes)])

    def b1(self):
        return nn.Sequential([
                nn.Conv(64, kernel_size=(7, 7), strides=(2, 2), padding='same
↪'),
                nn.relu,
                lambda x: nn.max_pool(x, window_shape=(3, 3), strides=(2, 2),
                                      padding='same')])
```

```
class GoogleNet(d2l.Classifier):
    def b1(self):
        if tab.selected('mxnet'):
            net = nn.Sequential()
            net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3,
                              activation='relu'),
                    nn.MaxPool2D(pool_size=3, strides=2, padding=1))
            return net
        if tab.selected('pytorch'):
            return nn.Sequential(
                nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        if tab.selected('tensorflow'):
            return tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(64, 7, strides=2, padding='same',
                                       activation='relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2,
                                          padding='same')])
```

2 つ目のモジュールは 2 つの畳み込み層を使う。まず 64 チャネルの 1×1 畳み込み層を適用し、その後にチャネル数を 3 倍にする 3×3 畳み込み層を続ける。これは Inception ブロックの第 2 ブランチに対応し、body の設計を完了する。この時点でチャネル数は 192 である。

```
@d2l.add_to_class(GoogleNet)
def b2(self):
    if tab.selected('mxnet'):
        net = nn.Sequential()
        net.add(nn.Conv2D(64, kernel_size=1, activation='relu'),
                nn.Conv2D(192, kernel_size=3, padding=1, activation='relu'),
                nn.MaxPool2D(pool_size=3, strides=2, padding=1))
        return net
    if tab.selected('pytorch'):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=1), nn.ReLU(),
            nn.LazyConv2d(192, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
    if tab.selected('tensorflow'):
        return tf.keras.Sequential([
            tf.keras.layers.Conv2D(64, 1, activation='relu'),
            tf.keras.layers.Conv2D(192, 3, padding='same', activation='relu'),
            tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same')])
    if tab.selected('jax'):
        return nn.Sequential([nn.Conv(64, kernel_size=(1, 1)),
                              nn.relu,
                              nn.Conv(192, kernel_size=(3, 3), padding='same'),
                              nn.relu,
                              lambda x: nn.max_pool(x, window_shape=(3, 3),
                                                    strides=(2, 2),
                                                    padding='same')])
```

3 つ目のモジュールは、2 つの完全な Inception ブロックを直列に接続する。最初の Inception ブロックの出力チャネル数は $64 + 128 + 32 + 32 = 256$ である。これは 4 つのブランチ間での出力チャネル数の比が $2 : 4 : 1 : 1$ であることに相当する。

これを実現するために、まず第 2 ブランチと第 3 ブランチで入力次元をそれぞれ $\frac{1}{2}$ と $\frac{1}{12}$ に削減し、結果としてそれぞれ 96 = 192/2 チャネルと 16 = 192/12 チャネルにする。

2 つ目の Inception ブロックの出力チャネル数は $128+192+96+64=480$ に増え、比は $128:192:96:64=4:6:3:2$ になる。前と同様に、第 2 チャネルと第 3 チャネルでは中間次元数を減らす必要がある。それぞれ $\frac{1}{2}$ と $\frac{1}{8}$ のスケールで十分であり、結果としてそれぞれ 128 チャネルと 32 チャネルになる。これは以下の `Inception` ブロックのコンストラクタ引数に反映されている。

```
@d2l.add_to_class(GoogleNet)
def b3(self):
    if tab.selected('mxnet'):
        net = nn.Sequential()
        net.add(Inception(64, (96, 128), (16, 32), 32),
                Inception(128, (128, 192), (32, 96), 64),
                nn.MaxPool2D(pool_size=3, strides=2, padding=1))
        return net
    if tab.selected('pytorch'):
        return nn.Sequential(Inception(64, (96, 128), (16, 32), 32),
                             Inception(128, (128, 192), (32, 96), 64),
                             nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
    if tab.selected('tensorflow'):
        return tf.keras.models.Sequential([
            Inception(64, (96, 128), (16, 32), 32),
            Inception(128, (128, 192), (32, 96), 64),
            tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same')])
    if tab.selected('jax'):
        return nn.Sequential([Inception(64, (96, 128), (16, 32), 32),
                              Inception(128, (128, 192), (32, 96), 64),
                              lambda x: nn.max_pool(x, window_shape=(3, 3),
                                                    strides=(2, 2),
                                                    padding='same')])
```

4 つ目のモジュールはより複雑である。5 つの Inception ブロックを直列に接続し、それぞれの出力チャネル数は 192+208+48+64 = 512、160+224+64+64 = 512、$128+256+64+64=512$、$112+288+64+64=528$、$256+320+128+128=832$ である。各ブランチに割り当てられるチャネル数は 3 つ目のモジュールと似ているが、具体的な値は異なる。第 2 ブランチの 3×3 畳み込み層が最も多くのチャネルを出力し、次に 1×1 畳み込み層のみを持つ第 1 ブランチ、5×5 畳み込み層を持つ第 3 ブランチ、そして 3×3 最大プーリング層を持つ第 4 ブランチが続く。第 2 ブランチと第 3 ブランチでは、まず比率に従ってチャネル数を削減する。これらの比率は Inception ブロックごとに少しずつ異なる。

```
@d2l.add_to_class(GoogleNet)
def b4(self):
    if tab.selected('mxnet'):
        net = nn.Sequential()
        net.add(Inception(192, (96, 208), (16, 48), 64),
                Inception(160, (112, 224), (24, 64), 64),
                Inception(128, (128, 256), (24, 64), 64),
```

(continues on next page)

(continued from previous page)

```
                Inception(112, (144, 288), (32, 64), 64),
                Inception(256, (160, 320), (32, 128), 128),
                nn.MaxPool2D(pool_size=3, strides=2, padding=1))
        return net
    if tab.selected('pytorch'):
        return nn.Sequential(Inception(192, (96, 208), (16, 48), 64),
                             Inception(160, (112, 224), (24, 64), 64),
                             Inception(128, (128, 256), (24, 64), 64),
                             Inception(112, (144, 288), (32, 64), 64),
                             Inception(256, (160, 320), (32, 128), 128),
                             nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
    if tab.selected('tensorflow'):
        return tf.keras.Sequential([
            Inception(192, (96, 208), (16, 48), 64),
            Inception(160, (112, 224), (24, 64), 64),
            Inception(128, (128, 256), (24, 64), 64),
            Inception(112, (144, 288), (32, 64), 64),
            Inception(256, (160, 320), (32, 128), 128),
            tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding='same')])
    if tab.selected('jax'):
        return nn.Sequential([Inception(192, (96, 208), (16, 48), 64),
                              Inception(160, (112, 224), (24, 64), 64),
                              Inception(128, (128, 256), (24, 64), 64),
                              Inception(112, (144, 288), (32, 64), 64),
                              Inception(256, (160, 320), (32, 128), 128),
                              lambda x: nn.max_pool(x, window_shape=(3, 3),
                                                    strides=(2, 2),
                                                    padding='same')])
```

5 つ目のモジュールは 2 つの Inception ブロックからなり、出力チャネル数はそれぞれ $256+320+128+128=832$ と $384+384+128+128=1024$ である。各ブランチに割り当てられるチャネル数は 3 つ目と 4 つ目のモジュールと同じであるが、具体的な値は異なる。なお、5 つ目のブロックの後には出力層が続く。このブロックでは NiN と同様に global average pooling 層を使って、各チャネルの高さと幅を 1 に変換する。最後に、出力を 2 次元配列に変換し、その後に全結合層を適用する。この全結合層の出力数はラベルクラス数である。

```
@d2l.add_to_class(GoogleNet)
def b5(self):
    if tab.selected('mxnet'):
        net = nn.Sequential()
        net.add(Inception(256, (160, 320), (32, 128), 128),
                Inception(384, (192, 384), (48, 128), 128),
                nn.GlobalAvgPool2D())
        return net
    if tab.selected('pytorch'):
        return nn.Sequential(Inception(256, (160, 320), (32, 128), 128),
                             Inception(384, (192, 384), (48, 128), 128),
                             nn.AdaptiveAvgPool2d((1,1)), nn.Flatten())
    if tab.selected('tensorflow'):
        return tf.keras.Sequential([
            Inception(256, (160, 320), (32, 128), 128),
```

(continues on next page)

(continued from previous page)

```
            Inception(384, (192, 384), (48, 128), 128),
            tf.keras.layers.GlobalAvgPool2D(),
            tf.keras.layers.Flatten()])
    if tab.selected('jax'):
        return nn.Sequential([Inception(256, (160, 320), (32, 128), 128),
                              Inception(384, (192, 384), (48, 128), 128),
                              # Flax does not provide a GlobalAvgPool2D layer
                              lambda x: nn.avg_pool(x,
                                                    window_shape=x.shape[1:3],
                                                    strides=x.shape[1:3],
                                                    padding='valid'),
                              lambda x: x.reshape((x.shape[0], -1))])
```

ここまでで b1 から b5 までのすべてのブロックを定義したので、あとはそれらをまとめて完全なネットワークに組み立てるだけである。

```
@d2l.add_to_class(GoogleNet)
def __init__(self, lr=0.1, num_classes=10):
    super(GoogleNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
        self.net.add(self.b1(), self.b2(), self.b3(), self.b4(), self.b5(),
                     nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.b1(), self.b2(), self.b3(), self.b4(),
                                 self.b5(), nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.Sequential([
            self.b1(), self.b2(), self.b3(), self.b4(), self.b5(),
            tf.keras.layers.Dense(num_classes)])
```

```
@d2l.add_to_class(GoogleNet)
def __init__(self, lr=0.1, num_classes=10):
    super(GoogleNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
        self.net.add(self.b1(), self.b2(), self.b3(), self.b4(), self.b5(),
                     nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.b1(), self.b2(), self.b3(), self.b4(),
                                 self.b5(), nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.Sequential([
            self.b1(), self.b2(), self.b3(), self.b4(), self.b5(),
            tf.keras.layers.Dense(num_classes)])
```

```
model = GoogleNet().layer_summary((1, 96, 96, 1))
```

```
Sequential output shape:     (1, 24, 24, 64)
Sequential output shape:     (1, 12, 12, 192)
Sequential output shape:     (1, 6, 6, 480)
Sequential output shape:     (1, 3, 3, 832)
Sequential output shape:     (1, 1024)
Dense output shape:  (1, 10)
```

```
@d2l.add_to_class(GoogleNet)
def __init__(self, lr=0.1, num_classes=10):
    super(GoogleNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
        self.net.add(self.b1(), self.b2(), self.b3(), self.b4(), self.b5(),
                     nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.b1(), self.b2(), self.b3(), self.b4(),
                                 self.b5(), nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.Sequential([
            self.b1(), self.b2(), self.b3(), self.b4(), self.b5(),
            tf.keras.layers.Dense(num_classes)])
```

GoogLeNet モデルは計算量の大きいモデルである。選択されたチャネル数、次元削減前のブロック数、チャネル間での容量配分の比率など、比較的恣意的なハイパーパラメータが非常に多いことに注意しよう。その多くは、GoogLeNet が導入された当時、ネットワーク定義や設計探索のための自動化ツールがまだ利用できなかったことに起因する。たとえば現在では、優れた深層学習フレームワークが入力テンソルの次元を自動的に推論できるのは当然だと考える。当時は、そのような設定の多くを実験者が明示的に指定しなければならず、活発な実験の速度をしばしば低下させていた。さらに、自動探索に必要なツールもまだ発展途上で、初期の実験は主として高コストな総当たり探索、遺伝的アルゴリズム、その他類似の戦略に頼っていた。

当面の変更として行うのは、入力の高さと幅を 224 から 96 に減らし、Fashion-MNIST で妥当な学習時間にすることだけである。これにより計算が簡単になる。各モジュール間で出力形状がどのように変化するかを見てみよう。

```
model = GoogleNet().layer_summary((1, 1, 96, 96))
```

```
Sequential output shape:     torch.Size([1, 64, 24, 24])
Sequential output shape:     torch.Size([1, 192, 12, 12])
Sequential output shape:     torch.Size([1, 480, 6, 6])
Sequential output shape:     torch.Size([1, 832, 3, 3])
```

(continues on next page)

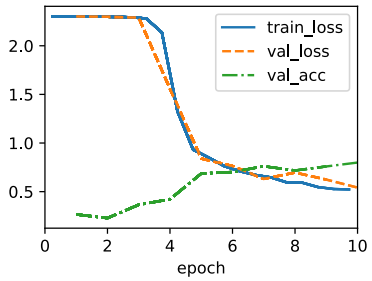
(continued from previous page)

```
Sequential output shape:     torch.Size([1, 1024])
Linear output shape:         torch.Size([1, 10])
```

```
model = GoogleNet().layer_summary((1, 1, 96, 96))
```

```
[07:44:36] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
Sequential output shape:     (1, 64, 24, 24)
Sequential output shape:     (1, 192, 12, 12)
Sequential output shape:     (1, 480, 6, 6)
Sequential output shape:     (1, 832, 3, 3)
Sequential output shape:     (1, 1024, 1, 1)
Dense output shape:  (1, 10)
```

```
model = GoogleNet(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



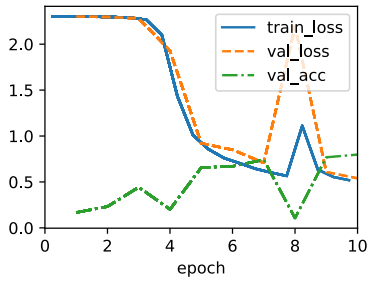

```
model = GoogleNet().layer_summary((1, 96, 96, 1))
```

```
Sequential output shape:     (1, 24, 24, 64)
Sequential output shape:     (1, 12, 12, 192)
Sequential output shape:     (1, 6, 6, 480)
Sequential output shape:     (1, 3, 3, 832)
Sequential output shape:     (1, 1024)
Dense output shape:  (1, 10)
```

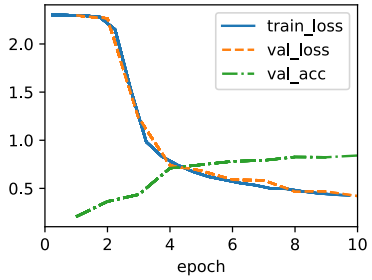
11.4.3 学習

これまでと同様に、Fashion-MNIST データセットを用いてモデルを学習する。学習手続きを呼び出す前に、画像を 96×96 ピクセルの解像度に変換する。

```
model = GoogleNet(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



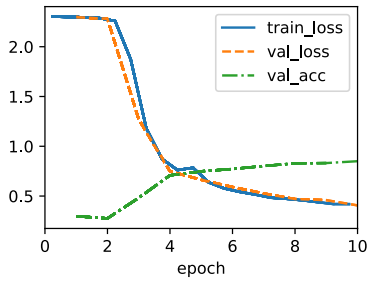

```
model = GoogleNet(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```




```
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
with d2l.try_gpu():
    model = GoogleNet(lr=0.01)
    trainer.fit(model, data)
```

11.4.4 議論

GoogLeNet の重要な特徴は、先行モデルよりも計算コストが低いにもかかわらず、精度が向上していることである。これは、ネットワークを評価するコストと誤差削減とのトレードオフを意識した、より慎重なネットワーク設計の始まりを示している。また、当時は完全に手作業ではあったものの、ブロック単位



でネットワーク設計のハイパーパラメータを試行する時代の始まりでもあった。ネットワーク構造の探索戦略を議論する Section 11.8 で、この話題に再び触れる。

以降の節では、batch normalization、残差接続、チャネルのグルーピングなど、ネットワークを大幅に改善できるいくつかの設計選択に出会う。今の時点では、おそらく最初の真に現代的な CNN を実装したことを誇りに思ってよいだろう。

11.4.5 演習

1. GoogLeNet は非常に成功したため、速度と精度を段階的に改善するいくつかの改良版が登場した。以下のものをいくつか実装して実行してみよ。
   1. Section 11.5 で後述するように、batch normalization 層 (Ioffe and Szegedy, 2015) を追加する。
   2. Szegedy *et al.* (2016) で述べられているように、Inception ブロックの調整（幅、畳み込みの選択と順序）を行う。
   3. Szegedy *et al.* (2016) で述べられているように、モデル正則化のために label smoothing を用いる。
   4. Section 11.6 で後述するように、残差接続 (Szegedy *et al.*, 2017) を追加して Inception ブロックをさらに調整する。
2. GoogLeNet が動作するために必要な最小の画像サイズはどれくらいか。
3. Fashion-MNIST のネイティブ解像度である 28×28 ピクセルで動作する GoogLeNet の変種を設計できるか？その場合、ネットワークの stem、body、head をどのように変更する必要があるか。あるいは、変更は不要だろうか？
4. AlexNet、VGG、NiN、GoogLeNet のモデルパラメータサイズを比較しよ。後者 2 つのネットワークアーキテクチャは、どのようにしてモデルパラメータサイズを大幅に削減しているのだろうか？
5. GoogLeNet と AlexNet に必要な計算量を比較しよ。これは、たとえばメモリ容量、メモリ帯域幅、キャッシュサイズ、計算量、特殊演算の利点といった観点で、アクセラレータチップの設計にどのような影響を与えるだろうか？

11.5 バッチ正規化

深層ニューラルネットワークの学習は難しいである。妥当な時間内に収束させるのは厄介なことがある。この節では、深層ネットワークの収束を一貫して加速する、広く使われている効果的な手法である バッチ正規化を説明する (Ioffe and Szegedy, 2015)。後ほど Section 11.6 で扱う残差ブロックと組み合わせることで、バッチ正規化により、実務者が 100 層を超えるネットワークを日常的に学習できるようになった。バッチ正規化には、副次的な（思いがけない）利点として、本質的な正則化効果もある。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import autograd, np, npx, init
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from functools import partial
from jax import numpy as jnp
import jax
import optax
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

11.5.1 深層ネットワークの学習

データを扱うとき、学習の前に前処理を行うことがよくある。データ前処理に関する選択は、最終結果に非常に大きな違いをもたらすことがある。家の価格を予測するために MLP を適用した例（Section 8.7）を思い出そう。実データを扱う際の最初のステップは、入力特徴量を複数の観測にわたって平均 0、分散 1 になるよう標準化することであった $\boldsymbol{\mu} = 0$ と $\boldsymbol{\Sigma} = 1$ (Friedman, 1987)。しばしば後者は対角成分が 1 になるように再スケーリングされ、すなわち $\Sigma_{ii} = 1$ とする。別の戦略として、ベクトルを単位長に再スケーリングし、場合によっては 各観測ごとに平均 0 にする方法もある。これは、たとえば空間センサーデータではうまく機能することがある。これらの前処理技術や他の多くの技術は、推定問題を適切に制御するうえで有益である。特徴選択と特徴抽出のレビューについては、たとえば Guyon *et al.* (2008) の論文を参照しよ。ベクトルを標準化することには、そこに作用する関数の複雑さを制約するという嬉しい副作用もある。たとえば、サポートベクターマシンにおける有名な radius-margin bound (Vapnik,

1995) やパーセプトロン収束定理 (Novikoff, 1962) は、ノルムが有界な入力に依存している。

直感的には、この標準化は最適化器と相性がよいである。というのも、パラメータを *a priori* に同程度のスケールに置くからである。そうであれば、深層ネットワークの 内部に対応する正規化のステップが有益でないかを問うのは自然である。これがバッチ正規化 (Ioffe and Szegedy, 2015) の発明に至った厳密な理由ではないが、バッチ正規化とその親戚である層正規化 (Ba *et al.*, 2016) を統一的な枠組みで理解するうえで有用な見方である。

第二に、典型的な MLP や CNN では、学習が進むにつれて、中間層の変数（たとえば MLP のアフィン変換の出力）は、入力から出力へ向かう層方向、同じ層内のユニット間、そしてモデルパラメータの更新による時間変化のために、大きさが大きく異なる値を取ることがある。バッチ正規化の発明者たちは、こうした変数の分布のドリフトがネットワークの収束を妨げる可能性があると、非公式に仮定した。直感的には、ある層の活性化が別の層の 100 倍の大きさで変動するなら、学習率に補償的な調整が必要になるかもしれない。AdaGrad (Duchi *et al.*, 2011)、Adam (Kingma and Ba, 2014)、Yogi (Zaheer *et al.*, 2018)、あるいは Distributed Shampoo (Anil *et al.*, 2020) のような適応的ソルバは、二次法の要素を加えるなどして、最適化の観点からこの問題に対処しようとする。別の方法は、単に適応的な正規化によって問題の発生を防ぐことである。

第三に、より深いネットワークは複雑であり、過学習しやすい傾向がある。これは、正則化がより重要になることを意味する。正則化の一般的な手法はノイズ注入である。これは昔から知られており、たとえば入力へのノイズ注入 (Bishop, 1995) に関しても研究されてきた。これは Section 8.6 におけるドロップアウトの基礎にもなっている。結果として、かなり幸運なことに、バッチ正規化はこの 3 つの利点すべて、すなわち前処理、数値安定性、正則化をもたらす。

バッチ正規化は個々の層に適用され、あるいは必要に応じてすべての層に適用される。各学習反復で、まず（バッチ正規化の）入力を平均を引き、標準偏差で割ることで正規化する。この平均と標準偏差は、現在のミニバッチの統計量に基づいて推定される。次に、失われた自由度を回復するためにスケール係数とオフセットを適用する。まさに バッチ統計量に基づく 正規化であることから、バッチ正規化という名前が付いている。

サイズ 1 のミニバッチでバッチ正規化を適用しようとしても、何も学習できないことに注意しよ。平均を引いた後、各隠れユニットの値は 0 になるからである。おそらく予想できるように、この節全体をバッチ正規化に割いているのは、十分に大きなミニバッチではこの手法が有効で安定だからである。ここから得られる教訓の一つは、バッチ正規化を適用する際には、バッチサイズの選択がバッチ正規化なしの場合よりもさらに重要であり、少なくとも、バッチサイズを調整するのと同様に適切な較正が必要だということである。

ミニバッチを $\mathcal{B}$ とし、$\mathbf{x} \in \mathcal{B}$ をバッチ正規化（BN）への入力とする。このと

き、バッチ正規化は次のように定義される。

$$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}. \tag{11.5.1}$$

(11.5.1) において、$\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ はミニバッチ $\mathcal{B}$ の標本平均であり、$\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ は標本標準偏差である。標準化を適用すると、得られるミニバッチは平均 0、分散 1 になる。分散 1 （他の任意の値ではなく）を選ぶのは恣意的である。この自由度は、$\mathbf{x}$ と同じ形状を持つ要素ごとの スケールパラメータ $\boldsymbol{\gamma}$ と シフトパラメータ $\boldsymbol{\beta}$ を含めることで回復する。どちらもモデル学習の一部として学習する必要があるパラメータである。

中間層の変数の大きさは、バッチ正規化が $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ と $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ を通じて、それらを所定の平均と大きさに積極的に中心化・再スケーリングするため、学習中に発散しません。特徴量の再スケーリングについて述べたときに触れたように、実践的な経験は、バッチ正規化によってより大きな学習率を使えることを示している。(11.5.1) における $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ と $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ は次のように計算する。

$$\hat{\boldsymbol{\mu}}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \text{ and } \hat{\boldsymbol{\sigma}}_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon. \tag{11.5.2}$$

分散推定値に小さな定数 $\epsilon > 0$ を加えることで、たとえ経験的分散推定値が非常に小さい、あるいは消失する場合でも、ゼロ除算を決して行わないようにしていることに注意しよ。$\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ と $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ の推定は、平均と分散のノイズを含む推定を用いることでスケーリングの問題に対処する。このノイズは問題だと思うかもしれない。しかし逆に、実際には有益である。

これは深層学習で繰り返し現れるテーマである。理論的にはまだ十分に特徴づけられていない理由により、最適化におけるさまざまなノイズ源は、しばしば学習の高速化と過学習の抑制につながる。この変動は正則化の一形態として働いているように見える。Teye *et al.* (2018) と Luo *et al.* (2018) は、バッチ正規化の性質をそれぞれベイズ事前分布とペナルティに関連付けた。特にこれは、なぜバッチ正規化が 50〜100 程度の中程度のミニバッチサイズで最もよく機能するのか、という謎に光を当てる。この特定のミニバッチサイズは、$\hat{\boldsymbol{\sigma}}$ によるスケールの面でも、$\hat{\boldsymbol{\mu}}$ によるオフセットの面でも、各層にちょうど「適切な量」のノイズを注入しているように見える。より大きなミニバッチでは推定が安定するため正則化効果が弱まり、逆に小さすぎるミニバッチでは分散が大きすぎて有用な信号が失われる。さらにこの方向を探究し、別種の前処理やフィルタリングを考えることで、他の有効な正則化手法が見つかるかもしれない。

学習済みモデルを固定すると、平均と分散の推定にはデータセット全体を使う方が望ましいと思うかもしれない。学習が終わった後で、同じ画像が属するバッチによって分類結果が変わるのはなぜだろうか。学習中は、すべてのデータ例に対する中間変数がモデルを更新するたびに変化するため、このような厳密な計算は不可能である。しかし、モデルが学習された後なら、各層の変数の平均と分散をデータセット全体に基づいて計算できる。実際、これはバッチ正規化を用いるモデルでは標準的な実践である。したがって、バッチ正規化層は 学習モード（ミニバッチ統計量で正規化）と 予測モード（データセット統計量で

正規化）で異なる振る舞いをする。この形では、ノイズが学習中にのみ注入される Section 8.6 のドロップアウト正則化の振る舞いによく似ている。

11.5.2 バッチ正規化層

全結合層と畳み込み層のバッチ正規化の実装は少し異なる。バッチ正規化と他の層との重要な違いの一つは、前者が一度にミニバッチ全体を対象に動作するため、他の層を導入したときのようにバッチ次元を無視できないことである。

全結合層

全結合層にバッチ正規化を適用する場合、Ioffe and Szegedy (2015) の元論文では、アフィン変換の後、非線形活性化関数の 前にバッチ正規化を挿入していた。後の応用では、活性化関数の直後にバッチ正規化を挿入する実験も行われた。全結合層への入力を $\mathbf{x}$、アフィン変換を $\mathbf{W}\mathbf{x}+\mathbf{b}$（重みパラメータを $\mathbf{W}$、バイアスパラメータを $\mathbf{b}$ とする）、活性化関数を ϕ とすると、バッチ正規化を用いた全結合層の出力 $\mathbf{h}$ は次のように表せる。

$$\mathbf{h} = \phi(\mathrm{BN}(\mathbf{W}\mathbf{x}+\mathbf{b})). \tag{11.5.3}$$

平均と分散は、変換を適用するのと 同じミニバッチ上で計算されることを思い出しよ。

畳み込み層

同様に、畳み込み層では、畳み込みの後、非線形活性化関数の前にバッチ正規化を適用できる。全結合層におけるバッチ正規化との重要な違いは、操作を各チャネルごとに すべての位置にわたって適用することである。これは、畳み込みを導いた位置不変性の仮定と整合的である。つまり、画像内のパターンの具体的な位置は、理解の目的にとって本質的ではないと仮定していた。

ミニバッチに m 個の例が含まれ、各チャネルについて、畳み込みの出力の高さが p、幅が q であるとする。畳み込み層では、各バッチ正規化を出力チャネルごとに $m \cdot p \cdot q$ 個の要素全体に対して同時に行う。したがって、平均と分散を計算するときにはすべての空間位置の値を集め、その結果、あるチャネル内では同じ平均と分散を各空間位置の値の正規化に適用する。各チャネルには独自のスケールパラメータとシフトパラメータがあり、どちらもスカラーである。

層正規化

畳み込みの文脈では、サイズ 1 のミニバッチでもバッチ正規化は十分に定義されることに注意しよ。結局のところ、画像全体の位置を平均に使えるからである。したがって、たとえ単一の観測内であっても、平均と分散は十分に定義される。この考察から Ba *et al.* (2016) は 層正規化の概念を導入した。これはバッ

チ正規化と同様に動作するが、1 つの観測に対して一度に適用される点だけが異なる。したがって、オフセットとスケーリング係数の両方はスカラーである。n 次元ベクトル $\mathbf{x}$ に対して、層正規化は次のように与えられる。

$$\mathbf{x} \rightarrow \mathrm{LN}(\mathbf{x}) = \frac{\mathbf{x} - \hat{\mu}}{\hat{\sigma}}, \tag{11.5.4}$$

ここでスケーリングとオフセットは係数ごとに適用され、次のように与えられる。

$$\hat{\mu} \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^{n} x_i \text{ and } \hat{\sigma}^2 \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2 + \epsilon. \tag{11.5.5}$$

前と同様に、ゼロ除算を防ぐために小さなオフセット $\epsilon > 0$ を加える。層正規化を使う大きな利点の一つは、発散を防ぐことである。結局のところ、ϵ を無視すれば、層正規化の出力はスケールに依存しない。すなわち、任意の $\alpha \neq 0$ に対して $\mathrm{LN}(\mathbf{x}) \approx \mathrm{LN}(\alpha\mathbf{x})$ が成り立つ。これは $|\alpha| \rightarrow \infty$ で等式になる（近似等式であるのは、分散に対するオフセット ϵ のためである）。

層正規化のもう一つの利点は、ミニバッチサイズに依存しないことである。また、学習時かテスト時かにも依存しません。言い換えると、これは単に、活性化を所定のスケールに標準化する決定論的な変換である。これは最適化における発散を防ぐうえで非常に有益である。ここではこれ以上の詳細は省き、興味のある読者には元論文を参照することを勧める。

予測時のバッチ正規化

前に述べたように、バッチ正規化は通常、学習モードと予測モードで異なる振る舞いをする。第一に、各ミニバッチで推定したときに生じる標本平均と標本分散のノイズは、モデルを学習し終えた後にはもはや望ましくない。第二に、バッチごとの正規化統計量を計算する余裕がない場合もある。たとえば、モデルを 1 件ずつ予測に適用しなければならないことがある。

通常、学習後には、変数統計量の安定した推定値を得るためにデータセット全体を使い、予測時にはそれらを固定する。したがって、バッチ正規化はテスト時と学習時で異なる振る舞いをする。ドロップアウトにも同様の特徴があることを思い出しよ。

11.5.3 スクラッチからの実装

バッチ正規化が実際にどのように機能するかを見るために、以下でスクラッチから実装する。

```
def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use is_grad_enabled to determine whether we are in training mode
    if not torch.is_grad_enabled():
        # In prediction mode, use mean and variance obtained by moving average
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
```

(continues on next page)

(continued from previous page)

```
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # When using a fully connected layer, calculate the mean and
            # variance on the feature dimension
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # When using a two-dimensional convolutional layer, calculate the
            # mean and variance on the channel dimension (axis=1). Here we
            # need to maintain the shape of X, so that the broadcasting
            # operation can be carried out later
            mean = X.mean(dim=(0, 2, 3), keepdim=True)
            var = ((X - mean) ** 2).mean(dim=(0, 2, 3), keepdim=True)
        # In training mode, the current mean and variance are used
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # Update the mean and variance using moving average
        moving_mean = (1.0 - momentum) * moving_mean + momentum * mean
        moving_var = (1.0 - momentum) * moving_var + momentum * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y, moving_mean.data, moving_var.data
```

```
def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use autograd to determine whether we are in training mode
    if not autograd.is_training():
        # In prediction mode, use mean and variance obtained by moving average
        X_hat = (X - moving_mean) / np.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # When using a fully connected layer, calculate the mean and
            # variance on the feature dimension
            mean = X.mean(axis=0)
            var = ((X - mean) ** 2).mean(axis=0)
        else:
            # When using a two-dimensional convolutional layer, calculate the
            # mean and variance on the channel dimension (axis=1). Here we
            # need to maintain the shape of X, so that the broadcasting
            # operation can be carried out later
            mean = X.mean(axis=(0, 2, 3), keepdims=True)
            var = ((X - mean) ** 2).mean(axis=(0, 2, 3), keepdims=True)
        # In training mode, the current mean and variance are used
        X_hat = (X - mean) / np.sqrt(var + eps)
        # Update the mean and variance using moving average
        moving_mean = (1.0 - momentum) * moving_mean + momentum * mean
        moving_var = (1.0 - momentum) * moving_var + momentum * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y, moving_mean, moving_var
```

```
def batch_norm(X, deterministic, gamma, beta, moving_mean, moving_var, eps,
               momentum):
    # Use `deterministic` to determine whether the current mode is training
    # mode or prediction mode
    if deterministic:
```

(continues on next page)

(continued from previous page)

```
        # In prediction mode, use mean and variance obtained by moving average
        # `linen.Module.variables` have a `value` attribute containing the␣
↪array
        X_hat = (X - moving_mean.value) / jnp.sqrt(moving_var.value + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # When using a fully connected layer, calculate the mean and
            # variance on the feature dimension
            mean = X.mean(axis=0)
            var = ((X - mean) ** 2).mean(axis=0)
        else:
            # When using a two-dimensional convolutional layer, calculate the
            # mean and variance on the channel dimension (axis=1). Here we
            # need to maintain the shape of X, so that the broadcasting
            # operation can be carried out later
            mean = X.mean(axis=(0, 2, 3), keepdims=True)
            var = ((X - mean) ** 2).mean(axis=(0, 2, 3), keepdims=True)
        # In training mode, the current mean and variance are used
        X_hat = (X - mean) / jnp.sqrt(var + eps)
        # Update the mean and variance using moving average
        moving_mean.value = momentum * moving_mean.value + (1.0 - momentum) *␣
↪mean
        moving_var.value = momentum * moving_var.value + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y
```

```
def batch_norm(X, gamma, beta, moving_mean, moving_var, eps):
    # Compute reciprocal of square root of the moving variance elementwise
    inv = tf.cast(tf.math.rsqrt(moving_var + eps), X.dtype)
    # Scale and shift
    inv *= gamma
    Y = X * inv + (beta - moving_mean * inv)
    return Y
```

これで適切な BatchNorm 層を作成できる。この層は、スケール gamma とシフト beta のための適切なパラメータを保持し、これらは学習の過程で更新される。さらに、この層はモデル予測時に後で使うための平均と分散の移動平均も保持する。

アルゴリズムの詳細はさておき、実装の背後にある設計パターンに注目しよう。通常、数学的な処理は batch_norm のような別関数で定義する。その後、この機能をカスタム層に統合し、そのコードは主に、データを適切なデバイスコンテキストへ移すこと、必要な変数を割り当てて初期化すること、移動平均（ここでは平均と分散）を追跡することなどの管理作業を担当する。このパターンにより、数学と定型コードをきれいに分離できる。また、便宜上ここでは入力形状を自動推論することを考慮していないため、特徴数を通して指定する必要がある。現在では、すべての主要な深層学習フレームワークが、高レベルのバッチ正規化 API でサイズと形状を自動検出できる（実際にはこちらを使う）。

```
class BatchNorm(nn.Module):
    # num_features: the number of outputs for a fully connected layer or the
    # number of output channels for a convolutional layer. num_dims: 2 for a
    # fully connected layer and 4 for a convolutional layer
    def __init__(self, num_features, num_dims):
        super().__init__()
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # The scale parameter and the shift parameter (model parameters) are
        # initialized to 1 and 0, respectively
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # The variables that are not model parameters are initialized to 0 and
        # 1
        self.moving_mean = torch.zeros(shape)
        self.moving_var = torch.ones(shape)

    def forward(self, X):
        # If X is not on the main memory, copy moving_mean and moving_var to
        # the device where X is located
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device)
            self.moving_var = self.moving_var.to(X.device)
        # Save the updated moving_mean and moving_var
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma, self.beta, self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.1)
        return Y
```

```
class BatchNorm(nn.Block):
    # `num_features`: the number of outputs for a fully connected layer
    # or the number of output channels for a convolutional layer. `num_dims`:
    # 2 for a fully connected layer and 4 for a convolutional layer
    def __init__(self, num_features, num_dims, **kwargs):
        super().__init__(**kwargs)
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # The scale parameter and the shift parameter (model parameters) are
        # initialized to 1 and 0, respectively
        self.gamma = self.params.get('gamma', shape=shape, init=init.One())
        self.beta = self.params.get('beta', shape=shape, init=init.Zero())
        # The variables that are not model parameters are initialized to 0 and
        # 1
        self.moving_mean = np.zeros(shape)
        self.moving_var = np.ones(shape)

    def forward(self, X):
        # If `X` is not on the main memory, copy `moving_mean` and
        # `moving_var` to the device where `X` is located
        if self.moving_mean.ctx != X.ctx:
            self.moving_mean = self.moving_mean.copyto(X.ctx)
```

(continues on next page)

(continued from previous page)

```
            self.moving_var = self.moving_var.copyto(X.ctx)
        # Save the updated `moving_mean` and `moving_var`
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma.data(), self.beta.data(), self.moving_mean,
            self.moving_var, eps=1e-12, momentum=0.1)
        return Y
```

```
class BatchNorm(nn.Module):
    # `num_features`: the number of outputs for a fully connected layer
    # or the number of output channels for a convolutional layer.
    # `num_dims`: 2 for a fully connected layer and 4 for a convolutional layer
    # Use `deterministic` to determine whether the current mode is training
    # mode or prediction mode
    num_features: int
    num_dims: int
    deterministic: bool = False

    @nn.compact
    def __call__(self, X):
        if self.num_dims == 2:
            shape = (1, self.num_features)
        else:
            shape = (1, 1, 1, self.num_features)

        # The scale parameter and the shift parameter (model parameters) are
        # initialized to 1 and 0, respectively
        gamma = self.param('gamma', jax.nn.initializers.ones, shape)
        beta = self.param('beta', jax.nn.initializers.zeros, shape)

        # The variables that are not model parameters are initialized to 0 and
        # 1. Save them to the 'batch_stats' collection
        moving_mean = self.variable('batch_stats', 'moving_mean', jnp.zeros,␣
↪shape)
        moving_var = self.variable('batch_stats', 'moving_var', jnp.ones,␣
↪shape)
        Y = batch_norm(X, self.deterministic, gamma, beta,
                       moving_mean, moving_var, eps=1e-5, momentum=0.9)

        return Y
```

```
class BatchNorm(tf.keras.layers.Layer):
    def __init__(self, **kwargs):
        super(BatchNorm, self).__init__(**kwargs)

    def build(self, input_shape):
        weight_shape = [input_shape[-1], ]
        # The scale parameter and the shift parameter (model parameters) are
        # initialized to 1 and 0, respectively
        self.gamma = self.add_weight(name='gamma', shape=weight_shape,
            initializer=tf.initializers.ones, trainable=True)
        self.beta = self.add_weight(name='beta', shape=weight_shape,
            initializer=tf.initializers.zeros, trainable=True)
        # The variables that are not model parameters are initialized to 0
```

(continues on next page)

(continued from previous page)

```
        self.moving_mean = self.add_weight(name='moving_mean',
            shape=weight_shape, initializer=tf.initializers.zeros,
            trainable=False)
        self.moving_variance = self.add_weight(name='moving_variance',
            shape=weight_shape, initializer=tf.initializers.ones,
            trainable=False)
        super(BatchNorm, self).build(input_shape)

    def assign_moving_average(self, variable, value):
        momentum = 0.1
        delta = (1.0 - momentum) * variable + momentum * value
        return variable.assign(delta)

    @tf.function
    def call(self, inputs, training):
        if training:
            axes = list(range(len(inputs.shape) - 1))
            batch_mean = tf.reduce_mean(inputs, axes, keepdims=True)
            batch_variance = tf.reduce_mean(tf.math.squared_difference(
                inputs, tf.stop_gradient(batch_mean)), axes, keepdims=True)
            batch_mean = tf.squeeze(batch_mean, axes)
            batch_variance = tf.squeeze(batch_variance, axes)
            mean_update = self.assign_moving_average(
                self.moving_mean, batch_mean)
            variance_update = self.assign_moving_average(
                self.moving_variance, batch_variance)
            self.add_update(mean_update)
            self.add_update(variance_update)
            mean, variance = batch_mean, batch_variance
        else:
            mean, variance = self.moving_mean, self.moving_variance
        output = batch_norm(inputs, moving_mean=mean, moving_var=variance,
            beta=self.beta, gamma=self.gamma, eps=1e-5)
        return output
```

momentum を用いて、過去の平均と分散の推定値をどの程度集約するかを制御した。これはやや紛らわしい名称で、最適化における モメンタム項とはまったく関係がない。それでも、この項に対して一般に採用されている名前であり、API の命名規則に敬意を払って、コードでも同じ変数名を使っている。

11.5.4 バッチ正規化付き LeNet

BatchNorm を文脈の中でどう適用するかを見るために、以下では従来の LeNet モデル（Section 10.6）に適用する。バッチ正規化は対応する活性化関数の前に、畳み込み層または全結合層の後に適用されることを思い出そう。

```
class BNLeNetScratch(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
```

(continues on next page)

(continued from previous page)

```
            self.net.add(
                nn.Conv2D(6, kernel_size=5), BatchNorm(6, num_dims=4),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Conv2D(16, kernel_size=5), BatchNorm(16, num_dims=4),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2), nn.Dense(120),
                BatchNorm(120, num_dims=2), nn.Activation('sigmoid'),
                nn.Dense(84), BatchNorm(84, num_dims=2),
                nn.Activation('sigmoid'), nn.Dense(num_classes))
            self.initialize()
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nn.LazyConv2d(6, kernel_size=5), BatchNorm(6, num_dims=4),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.LazyConv2d(16, kernel_size=5), BatchNorm(16, num_dims=4),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Flatten(), nn.LazyLinear(120),
                BatchNorm(120, num_dims=2), nn.Sigmoid(), nn.LazyLinear(84),
                BatchNorm(84, num_dims=2), nn.Sigmoid(),
                nn.LazyLinear(num_classes))
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=6, kernel_size=5,
                                       input_shape=(28, 28, 1)),
                BatchNorm(), tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Conv2D(filters=16, kernel_size=5),
                BatchNorm(), tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Flatten(), tf.keras.layers.Dense(120),
                BatchNorm(), tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(84), BatchNorm(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(num_classes)])
```

```
class BNLeNetScratch(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nn.Conv2D(6, kernel_size=5), BatchNorm(6, num_dims=4),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Conv2D(16, kernel_size=5), BatchNorm(16, num_dims=4),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2), nn.Dense(120),
                BatchNorm(120, num_dims=2), nn.Activation('sigmoid'),
                nn.Dense(84), BatchNorm(84, num_dims=2),
                nn.Activation('sigmoid'), nn.Dense(num_classes))
            self.initialize()
        if tab.selected('pytorch'):
```

(continues on next page)

(continued from previous page)

```
            self.net = nn.Sequential(
                nn.LazyConv2d(6, kernel_size=5), BatchNorm(6, num_dims=4),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.LazyConv2d(16, kernel_size=5), BatchNorm(16, num_dims=4),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Flatten(), nn.LazyLinear(120),
                BatchNorm(120, num_dims=2), nn.Sigmoid(), nn.LazyLinear(84),
                BatchNorm(84, num_dims=2), nn.Sigmoid(),
                nn.LazyLinear(num_classes))
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=6, kernel_size=5,
                                       input_shape=(28, 28, 1)),
                BatchNorm(), tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Conv2D(filters=16, kernel_size=5),
                BatchNorm(), tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Flatten(), tf.keras.layers.Dense(120),
                BatchNorm(), tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(84), BatchNorm(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(num_classes)])
```

```
class BNLeNetScratch(d2l.Classifier):
    lr: float = 0.1
    num_classes: int = 10
    training: bool = True

    def setup(self):
        self.net = nn.Sequential([
            nn.Conv(6, kernel_size=(5, 5)),
            BatchNorm(6, num_dims=4, deterministic=not self.training),
            nn.sigmoid,
            lambda x: nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2)),
            nn.Conv(16, kernel_size=(5, 5)),
            BatchNorm(16, num_dims=4, deterministic=not self.training),
            nn.sigmoid,
            lambda x: nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2)),
            lambda x: x.reshape((x.shape[0], -1)),
            nn.Dense(120),
            BatchNorm(120, num_dims=2, deterministic=not self.training),
            nn.sigmoid,
            nn.Dense(84),
            BatchNorm(84, num_dims=2, deterministic=not self.training),
            nn.sigmoid,
            nn.Dense(self.num_classes)])
```

```
class BNLeNetScratch(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
```

(continues on next page)

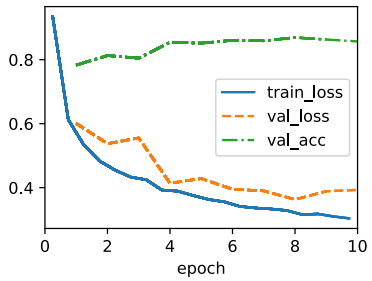
(continued from previous page)

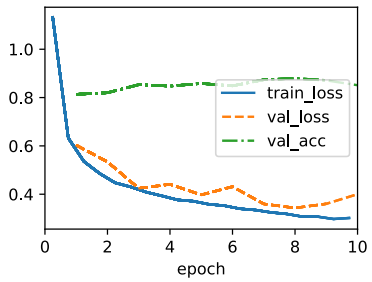
```
            self.net = nn.Sequential()
            self.net.add(
                nn.Conv2D(6, kernel_size=5), BatchNorm(6, num_dims=4),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Conv2D(16, kernel_size=5), BatchNorm(16, num_dims=4),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2), nn.Dense(120),
                BatchNorm(120, num_dims=2), nn.Activation('sigmoid'),
                nn.Dense(84), BatchNorm(84, num_dims=2),
                nn.Activation('sigmoid'), nn.Dense(num_classes))
            self.initialize()
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nn.LazyConv2d(6, kernel_size=5), BatchNorm(6, num_dims=4),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.LazyConv2d(16, kernel_size=5), BatchNorm(16, num_dims=4),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Flatten(), nn.LazyLinear(120),
                BatchNorm(120, num_dims=2), nn.Sigmoid(), nn.LazyLinear(84),
                BatchNorm(84, num_dims=2), nn.Sigmoid(),
                nn.LazyLinear(num_classes))
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=6, kernel_size=5,
                                       input_shape=(28, 28, 1)),
                BatchNorm(), tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Conv2D(filters=16, kernel_size=5),
                BatchNorm(), tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Flatten(), tf.keras.layers.Dense(120),
                BatchNorm(), tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(84), BatchNorm(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(num_classes)])
```

前と同様に、ネットワークを Fashion-MNIST データセットで学習する。このコードは、LeNet を最初に学習したときとほぼ同じである。

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = BNLeNetScratch(lr=0.1)
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = BNLeNetScratch(lr=0.1)
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```






```
@d2l.add_to_class(d2l.Classifier)  #@save
@partial(jax.jit, static_argnums=(0, 5))
def loss(self, params, X, Y, state, averaged=True):
    Y_hat, updates = state.apply_fn({'params': params,
                                     'batch_stats': state.batch_stats},
                                    *X, mutable=['batch_stats'],
                                    rngs={'dropout': state.dropout_rng})
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    Y = d2l.reshape(Y, (-1,))
    fn = optax.softmax_cross_entropy_with_integer_labels
    return (fn(Y_hat, Y).mean(), updates) if averaged else (fn(Y_hat, Y),␣
↪updates)
```

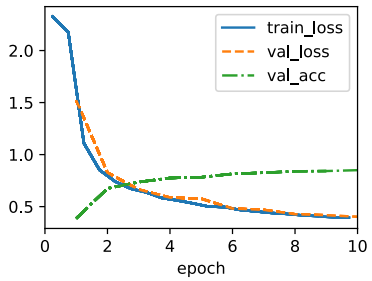
```
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=128)
with d2l.try_gpu():
    model = BNLeNetScratch(lr=0.5)
    trainer.fit(model, data)
```

最初のバッチ正規化層から学習されたスケールパラメータ gamma とシフトパラメータ beta を見てみよう。

```
model.net[1].gamma.reshape((-1,)), model.net[1].beta.reshape((-1,))
```

```
(tensor([2.0203, 1.5155, 1.8064, 2.0787, 1.9777, 1.3294], device='cuda:0',
        grad_fn=<ViewBackward0>),
 tensor([-1.0251, -0.6625,  1.7383,  1.5161, -0.3890, -0.2076], device='cuda:0
```

(continues on next page)



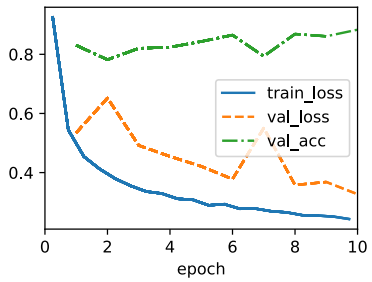
(continued from previous page)

```
↪',
       grad_fn=<ViewBackward0>))
```

```
model.net[1].gamma.data().reshape(-1,), model.net[1].beta.data().reshape(-1,)
```

```
(array([2.250208 , 1.5837682, 1.5654013, 1.962369 , 2.1671138, 1.4412792],␣
↪ctx=gpu(0)),
 array([ 1.3383483 ,  1.3885543 , -1.1205959 ,  0.24009949,  0.59068316,
        0.53491795], ctx=gpu(0)))
```

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = BNLeNetScratch(lr=0.1)
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```




```
tf.reshape(model.net.layers[1].gamma, (-1,)), tf.reshape(
    model.net.layers[1].beta, (-1,))
```

```
(<tf.Tensor: shape=(6,), dtype=float32, numpy=
 array([3.6470788, 3.568296 , 2.2124074, 3.306976 , 3.3767986, 3.3441424],
       dtype=float32)>,
```

(continues on next page)

(continued from previous page)

```
<tf.Tensor: shape=(6,), dtype=float32, numpy=
array([ 0.16249049,  0.06241638,  0.62027264,  0.11451854, -0.26635683,
       -0.01458476], dtype=float32)>)
```

11.5.5 簡潔な実装

先ほど自分で定義した BatchNorm クラスと比べると、深層学習フレームワークの高レベル API で定義された BatchNorm クラスを直接使う可能である。コードは上の実装とほぼ同じであるが、次元を正しく扱うための追加引数を与える必要がない点が異なる。

```
class BNLeNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nn.Conv2D(6, kernel_size=5), nn.BatchNorm(),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Conv2D(16, kernel_size=5), nn.BatchNorm(),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Dense(120), nn.BatchNorm(), nn.Activation('sigmoid'),
                nn.Dense(84), nn.BatchNorm(), nn.Activation('sigmoid'),
                nn.Dense(num_classes))
            self.initialize()
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nn.LazyConv2d(6, kernel_size=5), nn.LazyBatchNorm2d(),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.LazyConv2d(16, kernel_size=5), nn.LazyBatchNorm2d(),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Flatten(), nn.LazyLinear(120), nn.LazyBatchNorm1d(),
                nn.Sigmoid(), nn.LazyLinear(84), nn.LazyBatchNorm1d(),
                nn.Sigmoid(), nn.LazyLinear(num_classes))
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=6, kernel_size=5,
                                       input_shape=(28, 28, 1)),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Conv2D(filters=16, kernel_size=5),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Flatten(), tf.keras.layers.Dense(120),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(84),
```

(continues on next page)

(continued from previous page)

```
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(num_classes)])
```

```
class BNLeNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nn.Conv2D(6, kernel_size=5), nn.BatchNorm(),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Conv2D(16, kernel_size=5), nn.BatchNorm(),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Dense(120), nn.BatchNorm(), nn.Activation('sigmoid'),
                nn.Dense(84), nn.BatchNorm(), nn.Activation('sigmoid'),
                nn.Dense(num_classes))
            self.initialize()
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nn.LazyConv2d(6, kernel_size=5), nn.LazyBatchNorm2d(),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.LazyConv2d(16, kernel_size=5), nn.LazyBatchNorm2d(),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Flatten(), nn.LazyLinear(120), nn.LazyBatchNorm1d(),
                nn.Sigmoid(), nn.LazyLinear(84), nn.LazyBatchNorm1d(),
                nn.Sigmoid(), nn.LazyLinear(num_classes))
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=6, kernel_size=5,
                                       input_shape=(28, 28, 1)),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Conv2D(filters=16, kernel_size=5),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Flatten(), tf.keras.layers.Dense(120),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(84),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(num_classes)])
```

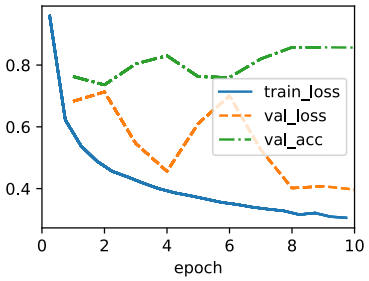
```
trainer.state.params['net']['layers_1']['gamma'].reshape((-1,)), \
trainer.state.params['net']['layers_1']['beta'].reshape((-1,))
```

```
(Array([1.5163397, 2.914868 , 1.4533901, 1.9001799, 2.4966698, 1.2752889],
↪ dtype=float32),
 Array([-0.11737688,  0.9210516 ,  0.3111974 , -0.49174237, -0.7915041 ,
        -0.16858716], dtype=float32))
```

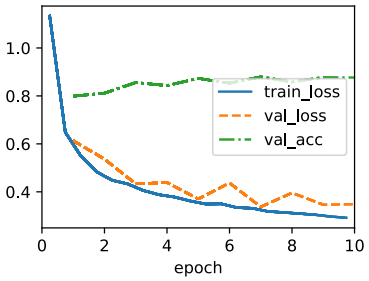
```
class BNLeNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.net = nn.Sequential()
            self.net.add(
                nn.Conv2D(6, kernel_size=5), nn.BatchNorm(),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Conv2D(16, kernel_size=5), nn.BatchNorm(),
                nn.Activation('sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Dense(120), nn.BatchNorm(), nn.Activation('sigmoid'),
                nn.Dense(84), nn.BatchNorm(), nn.Activation('sigmoid'),
                nn.Dense(num_classes))
            self.initialize()
        if tab.selected('pytorch'):
            self.net = nn.Sequential(
                nn.LazyConv2d(6, kernel_size=5), nn.LazyBatchNorm2d(),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.LazyConv2d(16, kernel_size=5), nn.LazyBatchNorm2d(),
                nn.Sigmoid(), nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Flatten(), nn.LazyLinear(120), nn.LazyBatchNorm1d(),
                nn.Sigmoid(), nn.LazyLinear(84), nn.LazyBatchNorm1d(),
                nn.Sigmoid(), nn.LazyLinear(num_classes))
        if tab.selected('tensorflow'):
            self.net = tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(filters=6, kernel_size=5,
                                       input_shape=(28, 28, 1)),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Conv2D(filters=16, kernel_size=5),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
                tf.keras.layers.Flatten(), tf.keras.layers.Dense(120),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(84),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('sigmoid'),
                tf.keras.layers.Dense(num_classes)])
```

以下では、同じハイパーパラメータを使ってモデルを学習する。通常どおり、高レベル API 版の方がはるかに高速に動作することに注意しよ。これは、そのコードが C++ や CUDA にコンパイルされている一方で、カスタム実装は Python によって解釈される必要があるためである。

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = BNLeNet(lr=0.1)
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```




```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = BNLeNet(lr=0.1)
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```




```
class BNLeNet(d2l.Classifier):
    lr: float = 0.1
    num_classes: int = 10
    training: bool = True

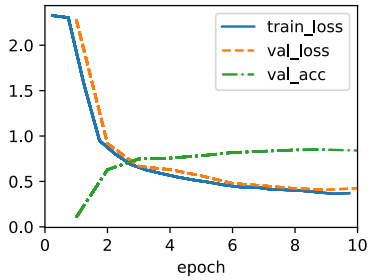
    def setup(self):
        self.net = nn.Sequential([
            nn.Conv(6, kernel_size=(5, 5)),
            nn.BatchNorm(not self.training),
            nn.sigmoid,
            lambda x: nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2)),
            nn.Conv(16, kernel_size=(5, 5)),
            nn.BatchNorm(not self.training),
            nn.sigmoid,
```

(continues on next page)

(continued from previous page)

```
            lambda x: nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2)),
            lambda x: x.reshape((x.shape[0], -1)),
            nn.Dense(120),
            nn.BatchNorm(not self.training),
            nn.sigmoid,
            nn.Dense(84),
            nn.BatchNorm(not self.training),
            nn.sigmoid,
            nn.Dense(self.num_classes)])
```

```
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=128)
with d2l.try_gpu():
    model = BNLeNet(lr=0.5)
    trainer.fit(model, data)
```



11.5.6 議論

直感的には、バッチ正規化は最適化の地形をより滑らかにすると考えられている。しかし、深層モデルの学習時に観測される現象については、推測的な直感と真の説明を区別しなければなりません。より単純な深層ニューラルネットワーク（MLP や従来型 CNN）が、そもそもなぜうまく汎化するのかすら、私たちはまだ知りません。ドロップアウトや重み減衰を加えても、それらは依然として非常に柔軟であり、未知データへの汎化能力には、より洗練された学習理論的な汎化保証が必要である可能性が高いである。

バッチ正規化を提案した元論文 (Ioffe and Szegedy, 2015) は、強力で有用な道具を導入しただけでなく、それがなぜ機能するのかについての説明も与えた。それは *internal covariate shift* を減らすからだ、という説明である。おそらく *internal covariate shift* とは、上で述べた直感、すなわち学習の過程で変数値の分布が変化するという考えを指していたのだろう。しかし、この説明には 2 つの問題があった。i) このドリフトは *covariate shift* とは大きく異なり、その名称は不適切である。むしろ concept drift に近いものである。ii) この説明は十分に規定されていない直感を与えるだけで、なぜこの手法が正確に機能するのかという問いは、厳密な説明を求める未解決のまま残す。本書全体を通して、実務者が深層ニューラルネットワークの開発を導くために用いる直感を伝えることを目指し

ている。しかし、こうした指針となる直感と、確立された科学的事実を区別することが重要だと考えている。やがてこの内容を習得し、自分で研究論文を書き始めるときには、技術的主張と勘や推測を明確に区別したいと思うはずである。

バッチ正規化の成功以降、*internal covariate shift* による説明は、技術文献や機械学習研究の提示方法をめぐるより広い議論の中で、繰り返し取り上げられてきた。2017 年の NeurIPS で Test of Time Award を受賞した際の印象的なスピーチで、Ali Rahimi は *internal covariate shift* を焦点として、現代の深層学習の実践を錬金術になぞらえる議論を展開した。その後、この例は、機械学習における懸念すべき傾向を概説したポジションペーパーで詳しく再検討された (Lipton and Steinhardt, 2018)。他の著者たちは、バッチ正規化の成功について別の説明を提案しており、その中には (Santurkar *et al.*, 2018) のように、バッチ正規化の成功は、元論文で主張されたものとはある意味で逆の振る舞いを示しているにもかかわらず達成されている、と主張するものもある。

internal covariate shift は、技術的な機械学習文献で毎年なされる何千もの同様に曖昧な主張と比べて、特に批判に値するわけではないことも指摘しておく。おそらく、この議論の焦点として強く響いたのは、対象読者にとって広く認識しやすかったからだろう。バッチ正規化は不可欠な手法であることが証明されており、実運用されている画像分類器のほぼすべてに適用されている。その結果、この手法を導入した論文は数万件もの引用を集めた。とはいえ、ノイズ注入による正則化、再スケーリングによる高速化、そして最後に前処理という指針は、将来さらに新しい層や手法の発明につながるだろうと私たちは考えている。

より実践的な観点から、バッチ正規化について覚えておくべき点がいくつかある。

- モデル学習中、バッチ正規化はミニバッチの平均と標準偏差を利用してネットワークの中間出力を継続的に調整し、ニューラルネットワーク全体の各層における中間出力の値をより安定させる。
- バッチ正規化は、全結合層と畳み込み層で少し異なる。実際、畳み込み層では、代わりに層正規化を使える場合がある。
- ドロップアウト層と同様に、バッチ正規化層は学習モードと予測モードで異なる振る舞いをする。
- バッチ正規化は、正則化と最適化の収束改善に有用である。対照的に、内部共変量シフトを減らすという元の動機は、妥当な説明ではないようである。
- 入力摂動に対してより頑健で、影響を受けにくいモデルを得たい場合は、バッチ正規化を取り除くことを検討してみよう (Wang *et al.*, 2022)。

11.5.7 演習

1. バッチ正規化の前に、全結合層または畳み込み層からバイアスパラメータを取り除くべきだろうか。なぜか。
2. バッチ正規化ありとなしの LeNet について、学習率を比較せよ。
    1. 検証精度の向上をプロットせよ。
    2. 最適化が失敗する前に、両方の場合で学習率をどこまで大きくできるか。
3. すべての層にバッチ正規化が必要だろうか。実験してみよう。
4. 平均だけを取り除く「lite」版のバッチ正規化、あるいは分散だけを取り除く版を実装せよ。どのように振る舞うか。
5. `beta` と `gamma` を固定せよ。結果を観察し、分析せよ。
6. ドロップアウトをバッチ正規化で置き換えられるか。振る舞いはどう変わるか。
7. 研究アイデア: 適用できそうな他の正規化変換を考えてみよう。
    1. 確率積分変換を適用できるか。
    2. フルランクの共分散推定を使えるか。なぜおそらく使うべきではないのだろうか。
    3. 他のコンパクトな行列変種（ブロック対角、低変位ランク、Monarch など）を使えるか。
    4. スパース化圧縮は正則化として働くか。
    5. 他の射影（たとえば凸錐、対称群固有の変換）を使えるか。

11.6 残差ネットワーク（ResNet）と ResNeXt

より深いネットワークを設計するにつれて、層を追加することでネットワークの複雑さと表現力がどのように増すのかを理解することが不可欠になる。さらに重要なのは、層を追加することで単に別のネットワークになるのではなく、ネットワークが厳密により表現力を持つように設計できるかどうかである。少し前進するために、少し数学が必要になる。

```
from d2l import torch as d2l
import torch
from torch import nn
from torch.nn import functional as F
```

```
from d2l import mxnet as d2l
from mxnet import np, npx, init
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from jax import numpy as jnp
import jax
```

```
import tensorflow as tf
from d2l import tensorflow as d2l
```

11.6.1 関数クラス

$\mathcal{F}$ を、特定のネットワークアーキテクチャ（学習率やその他のハイパーパラメータ設定も含む）が到達しうる関数のクラスと考えよう。つまり、すべての $f \in \mathcal{F}$ について、適切なデータセットでの学習によって得られるあるパラメータ集合（たとえば重みやバイアス）が存在する。本当に見つけたい「真の」関数を f^* としよう。それが $\mathcal{F}$ に含まれていれば理想的だが、通常はそこまで幸運ではない。その代わりに、$\mathcal{F}$ の中で最善と思われる $f^*_{\mathcal{F}}$ を見つけようとする。たとえば、特徴量 $\mathbf{X}$ とラベル $\mathbf{y}$ を持つデータセットが与えられたとき、次の最適化問題を解くことでそれを見つけようとするかもしれない：

$$f^*_{\mathcal{F}} \stackrel{\text{def}}{=} \operatorname*{argmin}_f L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}. \tag{11.6.1}$$

正則化 (Morozov, 1984, Tikhonov and Arsenin, 1977) は $\mathcal{F}$ の複雑さを制御し、整合性を達成できることが知られているので、学習データが多いほど一般により良い $f^*_{\mathcal{F}}$ が得られる。より強力な別のアーキテクチャ $\mathcal{F}'$ を設計すれば、より良い結果に到達できると考えるのはもっともである。言い換えれば、$f^*_{\mathcal{F}'}$ は $f^*_{\mathcal{F}}$ より「良い」と期待するだろう。しかし、$\mathcal{F} \not\subseteq \mathcal{F}'$ であれば、そのようなことが起こる保証はない。実際、$f^*_{\mathcal{F}'}$ のほうが悪いことも十分ありうる。Fig. 11.6.1 に示すように、入れ子になっていない関数クラスでは、より大きな関数クラスが必ずしも「真の」関数 f^* に近づくとは限らない。たとえば、Fig. 11.6.1 の左側では、$\mathcal{F}_3$ は $\mathcal{F}_1$ より f^* に近いが、$\mathcal{F}_6$ では逆に遠ざかっており、複雑さをさらに増やしても f^* からの距離が減る保証はない。一方、$\mathcal{F}_1 \subseteq \cdots \subseteq \mathcal{F}_6$ という入れ子の関数クラスでは、Fig. 11.6.1 の右側に示すように、前述の入れ子でない関数クラスの問題を避けることができる。

したがって、大きい関数クラスが小さい関数クラスを含む場合にのみ、関数クラスを大きくするとネットワークの表現力が厳密に増すことが保証される。深層ニューラルネットワークでは、新しく追加した層を恒等関数 $f(\mathbf{x}) = \mathbf{x}$ に学習できれば、新しいモデルは元のモデルと同等に有効である。新しいモデルが学習データによりよく適合する解を見つけられる可能性があるため、追加した層によって訓練誤差を下げやすくなるかもしれない。

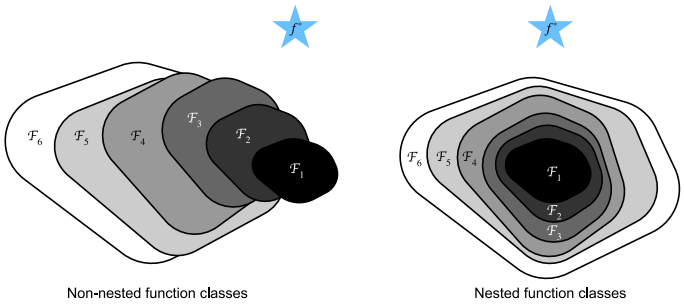


Fig. 11.6.1 入れ子になっていない関数クラスでは、より大きい（面積で示される）関数クラスが「真の」関数（f^*）に近づくことを保証しない。入れ子の関数クラスではこの問題は起こらない。

これは、非常に深いコンピュータビジョンモデルに取り組んでいた He *et al.* (2016) が考えた問題である。彼らが提案した 残差ネットワーク（*ResNet*）の核心は、追加される各層が恒等関数をその要素の一つとしてより容易に含められるようにする、という考え方にある。これらの考察はかなり深いが、驚くほど単純な解決策、すなわち 残差ブロックを導いた。これにより、ResNet は 2015 年の ImageNet Large Scale Visual Recognition Challenge で優勝した。この設計は、深層ニューラルネットワークの構築方法に大きな影響を与えた。たとえば、残差ブロックは再帰型ネットワークにも追加されている (Kim *et al.*, 2017, Prakash *et al.*, 2016)。同様に、Transformer (Vaswani *et al.*, 2017) では、多数の層を効率よく積み重ねるためにこれを用いている。これはグラフニューラルネットワークでも使われており (Kipf and Welling, 2016)、基本的な概念として、コンピュータビジョンでも広く用いられている (Redmon and Farhadi, 2018, Ren *et al.*, 2015)。なお、ResNet に先立って highway networks (Srivastava *et al.*, 2015) があり、恒等関数のまわりの洗練されたパラメータ化はないものの、いくつかの動機を共有している。

11.6.2 残差ブロック

Fig. 11.6.2 に示すように、ニューラルネットワークの局所的な部分に注目しよう。入力を $\mathbf{x}$ とする。学習によって得たい望ましい基礎写像を $f(\mathbf{x})$ とし、それが上部の活性化関数への入力として使われると仮定する。左側では、点線の枠内の部分が $f(\mathbf{x})$ を直接学習しなければならない。右側では、点線の枠内の部分が 残差写像 $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$ を学習する必要がある。これが残差ブロックの名前の由来である。恒等写像 $f(\mathbf{x}) = \mathbf{x}$ が望ましい基礎写像であるなら、残差写像は $g(\mathbf{x}) = 0$ となり、学習はより容易になる。つまり、点線の枠内にある上側の重み層（たとえば全結合層や畳み込み層）の重みとバイアスをゼロに押し込めばよい。右図は ResNet の 残差ブロックを示しており、層の入力 $\mathbf{x}$ を加算演算子へ運ぶ実線は 残差接続（または ショートカット接続）と呼ばれる。残差ブロックにより、入力は層をまたいで残差接続を通ってより速く順伝播できる。実際、残差ブロックはマルチブランチの Inception ブロックの特殊な場合とみなすこともできる。2 つの分岐を持ち、そのうちの一方が恒等写像である。

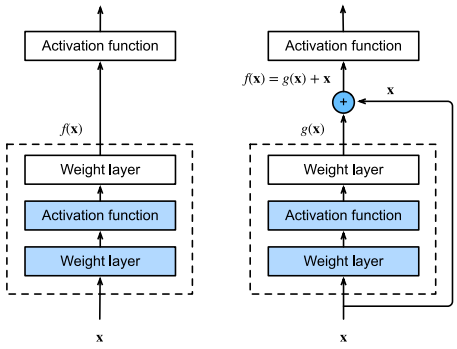


Fig. 11.6.2 通常のブロック（左）では、点線の枠内の部分が写像 $f(\mathbf{x})$ を直接学習しなければならない。残差ブロック（右）では、点線の枠内の部分が残差写像 $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$ を学習する必要があり、恒等写像 $f(\mathbf{x}) = \mathbf{x}$ を学習しやすくなる。

ResNet は VGG の 3×3 畳み込み層を全面的に採用した設計を持つ。残差ブロックは、同じ出力チャネル数を持つ 2 つの 3×3 畳み込み層からなる。各畳み込み層の後にはバッチ正規化層と ReLU 活性化関数が続く。その後、この 2 つの畳み込み演算を飛ばし、最後の ReLU 活性化関数の直前で入力を直接加える。この種の設計では、2 つの畳み込み層の出力が入力と同じ形状でなければならず、そうして初めて加算できる。チャネル数を変えたい場合は、追加の 1×1 畳み込み層を導入して、加算演算に必要な形状へ入力を変換する必要がある。以下のコードを見てみよう。

```
class Residual(nn.Module):  #@save
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

```
class Residual(nn.Block):  #@save
```

(continues on next page)

(continued from previous page)

```
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
        super().__init__(**kwargs)
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                               strides=strides)
        self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                   strides=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def forward(self, X):
        Y = npx.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return npx.relu(Y + X)
```

```
class Residual(nn.Module):  #@save
    """The Residual block of ResNet models."""
    num_channels: int
    use_1x1conv: bool = False
    strides: tuple = (1, 1)
    training: bool = True

    def setup(self):
        self.conv1 = nn.Conv(self.num_channels, kernel_size=(3, 3),
                             padding='same', strides=self.strides)
        self.conv2 = nn.Conv(self.num_channels, kernel_size=(3, 3),
                             padding='same')
        if self.use_1x1conv:
            self.conv3 = nn.Conv(self.num_channels, kernel_size=(1, 1),
                                 strides=self.strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm(not self.training)
        self.bn2 = nn.BatchNorm(not self.training)

    def __call__(self, X):
        Y = nn.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return nn.relu(Y)
```

```
class Residual(tf.keras.Model):  #@save
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        self.conv1 = tf.keras.layers.Conv2D(num_channels, padding='same',
                                            kernel_size=3, strides=strides)
        self.conv2 = tf.keras.layers.Conv2D(num_channels, kernel_size=3,
                                            padding='same')
        self.conv3 = None
        if use_1x1conv:
            self.conv3 = tf.keras.layers.Conv2D(num_channels, kernel_size=1,
                                                strides=strides)
        self.bn1 = tf.keras.layers.BatchNormalization()
        self.bn2 = tf.keras.layers.BatchNormalization()

    def call(self, X):
        Y = tf.keras.activations.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3 is not None:
            X = self.conv3(X)
        Y += X
        return tf.keras.activations.relu(Y)
```

このコードは 2 種類のネットワークを生成する。`use_1x1conv=False` のときは、ReLU 非線形性を適用する前に入力を出力へ加える。もう一方は、加算の前に 1×1 畳み込みによってチャネル数と解像度を調整する。Fig. 11.6.3 はこれを示している。

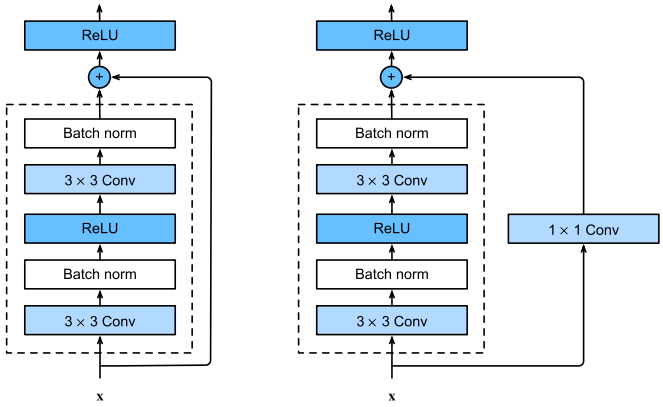


Fig. 11.6.3 1×1 畳み込みの有無による ResNet ブロック。1×1 畳み込みは、加算演算に必要な形状へ入力を変換する。

では、入力と出力が同じ形状である場合、つまり 1×1 畳み込みが不要な状況を見てみよう。

```
if tab.selected('mxnet'):
    blk = Residual(3)
    blk.initialize()
if tab.selected('pytorch'):
    blk = Residual(3)
```

(continues on next page)

(continued from previous page)

```
X = d2l.randn(4, 3, 6, 6)
blk(X).shape
```

```
if tab.selected('mxnet'):
    blk = Residual(3)
    blk.initialize()
if tab.selected('pytorch'):
    blk = Residual(3)
X = d2l.randn(4, 3, 6, 6)
blk(X).shape
```

```
blk = Residual(3)
X = jax.random.normal(d2l.get_key(), (4, 6, 6, 3))
blk.init_with_output(d2l.get_key(), X)[0].shape
```

```
(4, 6, 6, 3)
```

```
blk = Residual(3)
X = d2l.normal((4, 6, 6, 3))
Y = blk(X)
Y.shape
```

また、出力チャネル数を増やしながら出力の高さと幅を半分にすることもできる。この場合は `use_1x1conv=True` として 1×1 畳み込みを使う。これは各ResNetブロックの先頭で、`strides=2` によって空間次元を削減する際に便利である。

```
blk = Residual(6, use_1x1conv=True, strides=2)
if tab.selected('mxnet'):
    blk.initialize()
blk(X).shape
```

```
blk = Residual(6, use_1x1conv=True, strides=2)
if tab.selected('mxnet'):
    blk.initialize()
blk(X).shape
```

```
blk = Residual(6, use_1x1conv=True, strides=(2, 2))
blk.init_with_output(d2l.get_key(), X)[0].shape
```

```
(4, 3, 3, 6)
```

```
blk = Residual(6, use_1x1conv=True, strides=2)
```

(continues on next page)

(continued from previous page)

```
if tab.selected('mxnet'):
    blk.initialize()
blk(X).shape
```

11.6.3 ResNet モデル

ResNet の最初の 2 層は、先に説明した GoogLeNet と同じである。すなわち、64 個の出力チャネルを持ちストライド 2 の 7×7 畳み込み層の後に、ストライド 2 の 3×3 最大プーリング層が続く。違いは、ResNet では各畳み込み層の後にバッチ正規化層が追加されている点である。

```
class ResNet(d2l.Classifier):
    def b1(self):
        if tab.selected('mxnet'):
            net = nn.Sequential()
            net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
                    nn.BatchNorm(), nn.Activation('relu'),
                    nn.MaxPool2D(pool_size=3, strides=2, padding=1))
            return net
        if tab.selected('pytorch'):
            return nn.Sequential(
                nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                nn.LazyBatchNorm2d(), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        if tab.selected('tensorflow'):
            return tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(64, kernel_size=7, strides=2,
                                       padding='same'),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2,
                                          padding='same')])
```

```
class ResNet(d2l.Classifier):
    def b1(self):
        if tab.selected('mxnet'):
            net = nn.Sequential()
            net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
                    nn.BatchNorm(), nn.Activation('relu'),
                    nn.MaxPool2D(pool_size=3, strides=2, padding=1))
            return net
        if tab.selected('pytorch'):
            return nn.Sequential(
                nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                nn.LazyBatchNorm2d(), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        if tab.selected('tensorflow'):
            return tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(64, kernel_size=7, strides=2,
                                       padding='same'),
                tf.keras.layers.BatchNormalization(),
```

(continues on next page)

(continued from previous page)

```
                tf.keras.layers.Activation('relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2,
                                          padding='same')])
```

```
class ResNet(d2l.Classifier):
    arch: tuple
    lr: float = 0.1
    num_classes: int = 10
    training: bool = True

    def setup(self):
        self.net = self.create_net()

    def b1(self):
        return nn.Sequential([
            nn.Conv(64, kernel_size=(7, 7), strides=(2, 2), padding='same'),
            nn.BatchNorm(not self.training), nn.relu,
            lambda x: nn.max_pool(x, window_shape=(3, 3), strides=(2, 2),
                                  padding='same')])
```

```
class ResNet(d2l.Classifier):
    def b1(self):
        if tab.selected('mxnet'):
            net = nn.Sequential()
            net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
                    nn.BatchNorm(), nn.Activation('relu'),
                    nn.MaxPool2D(pool_size=3, strides=2, padding=1))
            return net
        if tab.selected('pytorch'):
            return nn.Sequential(
                nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                nn.LazyBatchNorm2d(), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        if tab.selected('tensorflow'):
            return tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(64, kernel_size=7, strides=2,
                                       padding='same'),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.Activation('relu'),
                tf.keras.layers.MaxPool2D(pool_size=3, strides=2,
                                          padding='same')])
```

GoogLeNet は Inception ブロックからなる 4 つのモジュールを使う。一方、ResNet は残差ブロックからなる 4 つのモジュールを使い、各モジュールは同じ出力チャネル数を持つ複数の残差ブロックで構成される。最初のモジュールのチャネル数は入力チャネル数と同じである。すでにストライド 2 の最大プーリング層が使われているので、高さと幅をさらに減らす必要はない。後続の各モジュールの最初の残差ブロックでは、前のモジュールに比べてチャネル数を 2 倍にし、高さと幅を半分にする。

```
@d2l.add_to_class(ResNet)
def block(self, num_residuals, num_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels))
    return nn.Sequential(*blk)
```

```
@d2l.add_to_class(ResNet)
def block(self, num_residuals, num_channels, first_block=False):
    blk = nn.Sequential()
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.add(Residual(num_channels))
    return blk
```

```
@d2l.add_to_class(ResNet)
def block(self, num_residuals, num_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(num_channels, use_1x1conv=True,
                                strides=(2, 2), training=self.training))
        else:
            blk.append(Residual(num_channels, training=self.training))
    return nn.Sequential(blk)
```

```
@d2l.add_to_class(ResNet)
def block(self, num_residuals, num_channels, first_block=False):
    blk = tf.keras.models.Sequential()
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.add(Residual(num_channels))
    return blk
```

次に、すべてのモジュールを ResNet に追加する。ここでは、各モジュールに 2 つの残差ブロックを使う。最後に、GoogLeNet と同様に、グローバル平均プーリング層の後に全結合層の出力を追加する。

```
@d2l.add_to_class(ResNet)
def __init__(self, arch, lr=0.1, num_classes=10):
    super(ResNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
```

(continues on next page)

(continued from previous page)

```
        self.net.add(self.b1())
        for i, b in enumerate(arch):
            self.net.add(self.block(*b, first_block=(i==0)))
        self.net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.b1())
        for i, b in enumerate(arch):
            self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))
        self.net.add_module('last', nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.models.Sequential(self.b1())
        for i, b in enumerate(arch):
            self.net.add(self.block(*b, first_block=(i==0)))
        self.net.add(tf.keras.models.Sequential([
            tf.keras.layers.GlobalAvgPool2D(),
            tf.keras.layers.Dense(units=num_classes)]))
```

```
@d2l.add_to_class(ResNet)
def __init__(self, arch, lr=0.1, num_classes=10):
    super(ResNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
        self.net.add(self.b1())
        for i, b in enumerate(arch):
            self.net.add(self.block(*b, first_block=(i==0)))
        self.net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.b1())
        for i, b in enumerate(arch):
            self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))
        self.net.add_module('last', nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.models.Sequential(self.b1())
        for i, b in enumerate(arch):
            self.net.add(self.block(*b, first_block=(i==0)))
        self.net.add(tf.keras.models.Sequential([
            tf.keras.layers.GlobalAvgPool2D(),
            tf.keras.layers.Dense(units=num_classes)]))
```

```
# %%tab jax
@d2l.add_to_class(ResNet)
def create_net(self):
    net = nn.Sequential([self.b1()])
    for i, b in enumerate(self.arch):
```

(continues on next page)

(continued from previous page)

```
        net.layers.extend([self.block(*b, first_block=(i==0))])
    net.layers.extend([nn.Sequential([
        # Flax does not provide a GlobalAvg2D layer
        lambda x: nn.avg_pool(x, window_shape=x.shape[1:3],
                              strides=x.shape[1:3], padding='valid'),
        lambda x: x.reshape((x.shape[0], -1)),
        nn.Dense(self.num_classes)])])
    return net
```

```
@d2l.add_to_class(ResNet)
def __init__(self, arch, lr=0.1, num_classes=10):
    super(ResNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
        self.net.add(self.b1())
        for i, b in enumerate(arch):
            self.net.add(self.block(*b, first_block=(i==0)))
        self.net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.b1())
        for i, b in enumerate(arch):
            self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))
        self.net.add_module('last', nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.models.Sequential(self.b1())
        for i, b in enumerate(arch):
            self.net.add(self.block(*b, first_block=(i==0)))
        self.net.add(tf.keras.models.Sequential([
            tf.keras.layers.GlobalAvgPool2D(),
            tf.keras.layers.Dense(units=num_classes)]))
```

```
# %%tab jax
@d2l.add_to_class(ResNet)
def create_net(self):
    net = nn.Sequential([self.b1()])
    for i, b in enumerate(self.arch):
        net.layers.extend([self.block(*b, first_block=(i==0))])
    net.layers.extend([nn.Sequential([
        # Flax does not provide a GlobalAvg2D layer
        lambda x: nn.avg_pool(x, window_shape=x.shape[1:3],
                              strides=x.shape[1:3], padding='valid'),
        lambda x: x.reshape((x.shape[0], -1)),
        nn.Dense(self.num_classes)])])
    return net
```

```
# %%tab jax
@d2l.add_to_class(ResNet)
```

(continues on next page)

(continued from previous page)

```
def create_net(self):
    net = nn.Sequential([self.b1()])
    for i, b in enumerate(self.arch):
        net.layers.extend([self.block(*b, first_block=(i==0))])
    net.layers.extend([nn.Sequential([
        # Flax does not provide a GlobalAvg2D layer
        lambda x: nn.avg_pool(x, window_shape=x.shape[1:3],
                              strides=x.shape[1:3], padding='valid'),
        lambda x: x.reshape((x.shape[0], -1)),
        nn.Dense(self.num_classes)])])
    return net
```

```
class ResNet18(ResNet):
    arch: tuple = ((2, 64), (2, 128), (2, 256), (2, 512))
    lr: float = 0.1
    num_classes: int = 10
```

```
# %%tab jax
@d2l.add_to_class(ResNet)
def create_net(self):
    net = nn.Sequential([self.b1()])
    for i, b in enumerate(self.arch):
        net.layers.extend([self.block(*b, first_block=(i==0))])
    net.layers.extend([nn.Sequential([
        # Flax does not provide a GlobalAvg2D layer
        lambda x: nn.avg_pool(x, window_shape=x.shape[1:3],
                              strides=x.shape[1:3], padding='valid'),
        lambda x: x.reshape((x.shape[0], -1)),
        nn.Dense(self.num_classes)])])
    return net
```

各モジュールには 4 つの畳み込み層がある（1×1 畳み込み層を除く）。最初の 7×7 畳み込み層と最後の全結合層を合わせると、合計 18 層になる。したがって、このモデルは一般に ResNet-18 と呼ばれる。モジュール内のチャネル数や残差ブロック数を変えることで、より深い 152 層の ResNet-152 のような別の ResNet モデルを作ることができる。ResNet の主なアーキテクチャは GoogLeNet に似ているが、ResNet の構造はより単純で修正しやすい。こうした要因により、ResNet は急速かつ広範に使われるようになった。Fig. 11.6.4 は ResNet-18 全体を示している。

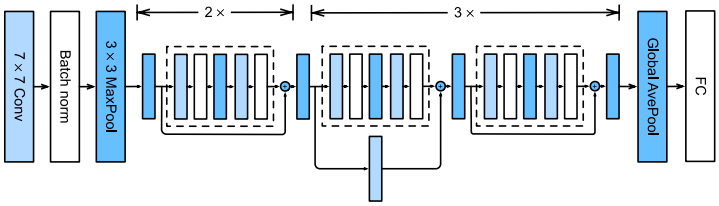


Fig. 11.6.4 ResNet-18 のアーキテクチャ。

ResNet を訓練する前に、ResNet の異なるモジュール間で入力形状がどのように変化するかを観察しよう。これまでのすべてのアーキテクチャと同様に、グローバル平均プーリング層がすべての特徴を集約する地点までは、解像度が下がる一方でチャネル数は増えていく。

```
class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                         lr, num_classes)
```

```
class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                         lr, num_classes)
```

```
ResNet18(training=False).layer_summary((1, 96, 96, 1))
```

```
Sequential output shape:     (1, 24, 24, 64)
Sequential output shape:     (1, 24, 24, 64)
Sequential output shape:     (1, 12, 12, 128)
Sequential output shape:     (1, 6, 6, 256)
Sequential output shape:     (1, 3, 3, 512)
Sequential output shape:     (1, 10)
```

```
class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                         lr, num_classes)
```

```
ResNet18().layer_summary((1, 1, 96, 96))
```

```
Sequential output shape:     torch.Size([1, 64, 24, 24])
Sequential output shape:     torch.Size([1, 64, 24, 24])
Sequential output shape:     torch.Size([1, 128, 12, 12])
Sequential output shape:     torch.Size([1, 256, 6, 6])
Sequential output shape:     torch.Size([1, 512, 3, 3])
Sequential output shape:     torch.Size([1, 10])
```

```
ResNet18().layer_summary((1, 1, 96, 96))
```

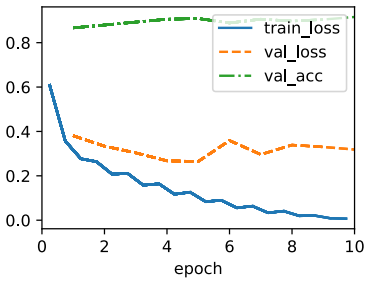
```
Sequential output shape:     (1, 64, 24, 24)
Sequential output shape:     (1, 64, 24, 24)
Sequential output shape:     (1, 128, 12, 12)
Sequential output shape:     (1, 256, 6, 6)
Sequential output shape:     (1, 512, 3, 3)
```

(continues on next page)

(continued from previous page)

```
GlobalAvgPool2D output shape:        (1, 512, 1, 1)
Dense output shape:  (1, 10)
```

```
model = ResNet18(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```




```
ResNet18().layer_summary((1, 96, 96, 1))
```

```
Sequential output shape:     (1, 24, 24, 64)
Sequential output shape:     (1, 24, 24, 64)
Sequential output shape:     (1, 12, 12, 128)
Sequential output shape:     (1, 6, 6, 256)
Sequential output shape:     (1, 3, 3, 512)
Sequential output shape:     (1, 10)
```

11.6.4 訓練

これまでと同様に、Fashion-MNIST データセットで ResNet を訓練する。ResNet は非常に強力で柔軟なアーキテクチャである。訓練損失と検証損失を示すグラフには、訓練損失がかなり低いという大きな差が見られる。この柔軟なネットワークでは、より多くの訓練データがあれば、その差を埋めて精度を改善するうえで明確な利点があるだろう。

```
model = ResNet18(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```

```
model = ResNet18(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
if tab.selected('pytorch'):
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```

```
class ResNeXtBlock(nn.Module):  #@save
    """The ResNeXt block."""
    num_channels: int
    groups: int
    bot_mul: int
    use_1x1conv: bool = False
    strides: tuple = (1, 1)
    training: bool = True

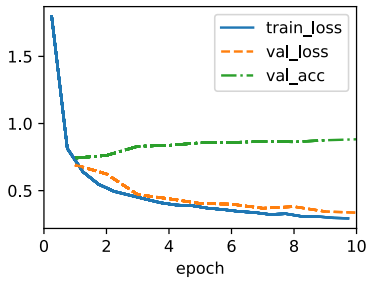
    def setup(self):
        bot_channels = int(round(self.num_channels * self.bot_mul))
        self.conv1 = nn.Conv(bot_channels, kernel_size=(1, 1),
                             strides=(1, 1))
        self.conv2 = nn.Conv(bot_channels, kernel_size=(3, 3),
                             strides=self.strides, padding='same',
                             feature_group_count=bot_channels//self.groups)
        self.conv3 = nn.Conv(self.num_channels, kernel_size=(1, 1),
                             strides=(1, 1))
        self.bn1 = nn.BatchNorm(not self.training)
        self.bn2 = nn.BatchNorm(not self.training)
        self.bn3 = nn.BatchNorm(not self.training)
        if self.use_1x1conv:
            self.conv4 = nn.Conv(self.num_channels, kernel_size=(1, 1),
                                 strides=self.strides)
            self.bn4 = nn.BatchNorm(not self.training)
        else:
            self.conv4 = None

    def __call__(self, X):
        Y = nn.relu(self.bn1(self.conv1(X)))
        Y = nn.relu(self.bn2(self.conv2(Y)))
        Y = self.bn3(self.conv3(Y))
        if self.conv4:
            X = self.bn4(self.conv4(X))
        return nn.relu(Y + X)
```

```
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
with d2l.try_gpu():
    model = ResNet18(lr=0.01)
    trainer.fit(model, data)
```

11.6.5 ResNeXt

ResNet の設計で直面する課題の一つは、あるブロック内で非線形性と次元数の間にあるトレードオフである。つまり、層数を増やすことで、あるいは畳み



込みの幅を増やすことで、より多くの非線形性を加えられる。別の戦略としては、ブロック間で情報を運ぶチャネル数を増やす方法がある。しかし後者には二次的なコストが伴う。なぜなら、c_i チャネルを受け取り c_o チャネルを出力する計算コストは $O(c_\mathrm{i} \cdot c_\mathrm{o})$ に比例するからである（Section 10.4 の議論を参照）。

Fig. 11.4.1 の Inception ブロックからも着想を得ることができる。そこでは情報がブロック内を複数の独立したグループに分かれて流れている。この複数の独立グループという考え方を Fig. 11.6.3 の ResNet ブロックに適用した結果、ResNeXt の設計が生まれた (Xie *et al.*, 2017)。Inception のように多種多様な変換を使うのではなく、ResNeXt はすべての分岐で 同じ変換を採用し、各分岐を個別に調整する必要を最小限にしている。

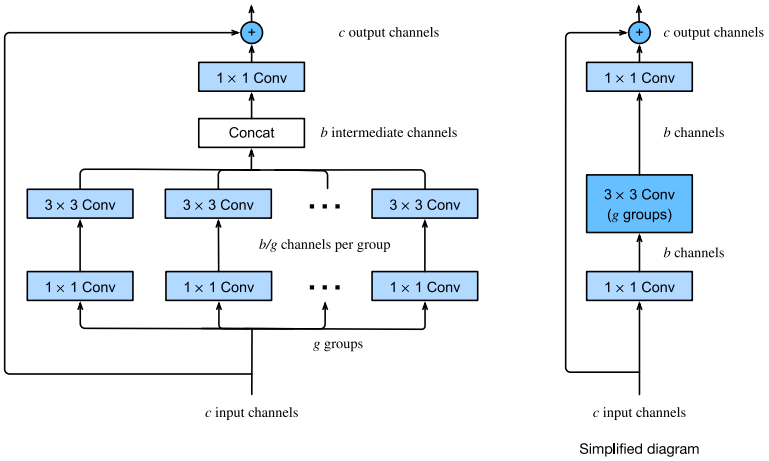


Fig. 11.6.5 ResNeXt ブロック。g 個のグループを持つ grouped convolution は、密な畳み込みよりも g 倍高速である。中間チャネル数 b が c より小さいとき、これはボトルネック残差ブロックである。

c_i チャネルから c_o チャネルへの畳み込みを、サイズ c_i/g の g 個のグループに分け、それぞれがサイズ c_o/g の g 個の出力を生成するように分割することを、まさにその名の通り *grouped convolution* と呼ぶ。計算コスト（比例）は $O(c_\mathrm{i} \cdot c_\mathrm{o})$ から $O(g \cdot (c_\mathrm{i}/g) \cdot (c_\mathrm{o}/g)) = O(c_\mathrm{i} \cdot c_\mathrm{o}/g)$ に減り、つまり g 倍高速になる。さらに良いことに、出力を生成するのに必要なパラメータ数も、$c_\mathrm{i} \times c_\mathrm{o}$ の行列から、サイ

ズ $(c_\text{i}/g) \times (c_\text{o}/g)$ のより小さな行列 g 個に減り、これもまた g 倍の削減になる。以下では、c_i と c_o の両方が g で割り切れると仮定する。

この設計における唯一の課題は、g 個のグループ間で情報が交換されないことである。Fig. 11.6.5 の ResNeXt ブロックは、これを 2 つの方法で補っている。すなわち、3×3 カーネルを持つ grouped convolution を 2 つの 1×1 畳み込みの間に挟んでいる。2 つ目の 1×1 畳み込みは、チャネル数を元に戻す役割も兼ねる。利点は、1×1 カーネルについては $O(c \cdot b)$ のコストしか払わず、3×3 カーネルについては $O(b^2/g)$ のコストで済むことである。Section 11.6.2 の残差ブロック実装と同様に、残差接続は 1×1 畳み込みに置き換えられる（したがって一般化される）。

Fig. 11.6.5 の右図は、得られるネットワークブロックをより簡潔にまとめている。これは、Section 11.8 における汎用的な現代 CNN の設計でも重要な役割を果たす。grouped convolution の考え方は AlexNet の実装にまでさかのぼることに注意しよう (Krizhevsky *et al.*, 2012)。メモリに制約のある 2 台の GPU にネットワークを分散させる際、実装では各 GPU を独立したチャネルとして扱っていたが、それで悪影響はなかった。

以下の `ResNeXtBlock` クラスの実装では、引数として `groups`（g）と、中間（ボトルネック）チャネル数 `bot_channels`（b）を取る。最後に、表現の高さと幅を減らす必要があるときは、`use_1x1conv=True, strides=2` としてストライド 2 を追加する。

```
class ResNeXtBlock(nn.Module):  #@save
    """The ResNeXt block."""
    def __init__(self, num_channels, groups, bot_mul, use_1x1conv=False,
                 strides=1):
        super().__init__()
        bot_channels = int(round(num_channels * bot_mul))
        self.conv1 = nn.LazyConv2d(bot_channels, kernel_size=1, stride=1)
        self.conv2 = nn.LazyConv2d(bot_channels, kernel_size=3,
                                   stride=strides, padding=1,
                                   groups=bot_channels//groups)
        self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1, stride=1)
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()
        self.bn3 = nn.LazyBatchNorm2d()
        if use_1x1conv:
            self.conv4 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides)
            self.bn4 = nn.LazyBatchNorm2d()
        else:
            self.conv4 = None

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = F.relu(self.bn2(self.conv2(Y)))
        Y = self.bn3(self.conv3(Y))
        if self.conv4:
            X = self.bn4(self.conv4(X))
        return F.relu(Y + X)
```

```
class ResNeXtBlock(nn.Block):  #@save
    """The ResNeXt block."""
    def __init__(self, num_channels, groups, bot_mul,
                 use_1x1conv=False, strides=1, **kwargs):
        super().__init__(**kwargs)
        bot_channels = int(round(num_channels * bot_mul))
        self.conv1 = nn.Conv2D(bot_channels, kernel_size=1, padding=0,
                               strides=1)
        self.conv2 = nn.Conv2D(bot_channels, kernel_size=3, padding=1,
                               strides=strides, groups=bot_channels//groups)
        self.conv3 = nn.Conv2D(num_channels, kernel_size=1, padding=0,
                               strides=1)
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()
        self.bn3 = nn.BatchNorm()
        if use_1x1conv:
            self.conv4 = nn.Conv2D(num_channels, kernel_size=1,
                                   strides=strides)
            self.bn4 = nn.BatchNorm()
        else:
            self.conv4 = None

    def forward(self, X):
        Y = npx.relu(self.bn1(self.conv1(X)))
        Y = npx.relu(self.bn2(self.conv2(Y)))
        Y = self.bn3(self.conv3(Y))
        if self.conv4:
            X = self.bn4(self.conv4(X))
        return npx.relu(Y + X)
```

```
blk = ResNeXtBlock(32, 16, 1)
X = jnp.zeros((4, 96, 96, 32))
blk.init_with_output(d2l.get_key(), X)[0].shape
```

```
(4, 96, 96, 32)
```

```
class ResNeXtBlock(tf.keras.Model):  #@save
    """The ResNeXt block."""
    def __init__(self, num_channels, groups, bot_mul, use_1x1conv=False,
                 strides=1):
        super().__init__()
        bot_channels = int(round(num_channels * bot_mul))
        self.conv1 = tf.keras.layers.Conv2D(bot_channels, 1, strides=1)
        self.conv2 = tf.keras.layers.Conv2D(bot_channels, 3, strides=strides,
                                            padding="same",
                                            groups=bot_channels//groups)
        self.conv3 = tf.keras.layers.Conv2D(num_channels, 1, strides=1)
        self.bn1 = tf.keras.layers.BatchNormalization()
        self.bn2 = tf.keras.layers.BatchNormalization()
        self.bn3 = tf.keras.layers.BatchNormalization()
        if use_1x1conv:
            self.conv4 = tf.keras.layers.Conv2D(num_channels, 1,
```

(continues on next page)

(continued from previous page)

```
                                                strides=strides)
            self.bn4 = tf.keras.layers.BatchNormalization()
        else:
            self.conv4 = None

    def call(self, X):
        Y = tf.keras.activations.relu(self.bn1(self.conv1(X)))
        Y = tf.keras.activations.relu(self.bn2(self.conv2(Y)))
        Y = self.bn3(self.conv3(Y))
        if self.conv4:
            X = self.bn4(self.conv4(X))
        return tf.keras.activations.relu(Y + X)
```

その使い方は、先に述べた ResNetBlock とまったく同様である。たとえば、(use_1x1conv=False, strides=1) を使うと、入力と出力は同じ形状になる。あるいは、use_1x1conv=True, strides=2 とすると、出力の高さと幅が半分になる。

```
blk = ResNeXtBlock(32, 16, 1)
if tab.selected('mxnet'):
    blk.initialize()
X = d2l.randn(4, 32, 96, 96)
blk(X).shape
```

```
blk = ResNeXtBlock(32, 16, 1)
if tab.selected('mxnet'):
    blk.initialize()
X = d2l.randn(4, 32, 96, 96)
blk(X).shape
```

```
blk = ResNeXtBlock(32, 16, 1)
X = d2l.normal((4, 96, 96, 32))
Y = blk(X)
Y.shape
```

11.6.6 まとめと考察

入れ子の関数クラスは望ましい。なぜなら、容量を増やしたときに、単により強力な関数クラスを得られるのであって、微妙に 異なる関数クラスになってしまうのを避けられるからである。これを実現する一つの方法は、追加した層が入力をそのまま出力へ通すようにすることである。残差接続はこれを可能にする。その結果、単純な関数の形が $f(\mathbf{x}) = 0$ であるという帰納バイアスから、単純な関数が $f(\mathbf{x}) = \mathbf{x}$ のように見えるという帰納バイアスへと変わる。

残差写像は、重み層のパラメータをゼロに押し込むような形で、恒等関数をより容易に学習できる。残差ブロックを用いることで、有効な 深いニューラルネットワークを訓練できる。入力は層をまたいで残差接続を通ってより速く順伝播できる。その結果、はるかに深いネットワークを訓練できる。たとえば、元

の ResNet 論文 (He *et al.*, 2016) では最大 152 層まで可能になった。残差ネットワークのもう一つの利点は、訓練の途中で恒等関数として初期化された層を追加できることである。結局のところ、層のデフォルトの振る舞いは、データを変えずに通すことだからである。これは場合によっては、非常に大きなネットワークの訓練を加速できる。

残差接続が登場する以前には、ゲーティングユニットを持つバイパス経路が導入され、100 層を超える highway networks を効果的に訓練できるようにした (Srivastava *et al.*, 2015)。恒等関数をバイパス経路として用いることで、ResNet は複数のコンピュータビジョンタスクで非常に優れた性能を示した。残差接続は、その後の畳み込み型あるいは系列型の深層ニューラルネットワーク設計に大きな影響を与えた。後で紹介するように、Transformer アーキテクチャ (Vaswani *et al.*, 2017) は残差接続（および他の設計選択）を採用しており、言語、画像、音声、強化学習など多様な分野に広く浸透している。

ResNeXt は、畳み込みニューラルネットワークの設計が時間とともにどのように進化してきたかを示す一例である。計算をより節約し、その代わりに活性化のサイズ（チャネル数）を調整することで、より低コストでより高速かつ高精度なネットワークを実現できる。grouped convolution を別の見方で捉えると、畳み込み重みのブロック対角行列を考えることができる。なお、より効率的なネットワークにつながるこの種の「工夫」はかなり多い。たとえば ShiftNet (Wu *et al.*, 2018) は、チャネルにシフトした活性化を加えるだけで 3×3 畳み込みの効果を模倣し、計算コストを増やさずに関数の複雑さを高めている。

ここまで議論してきた設計に共通する特徴は、ネットワーク設計がかなり手作業に依存しており、主として設計者の創意工夫に頼って「正しい」ネットワークハイパーパラメータを見つける点である。これは明らかに実現可能ではあるが、人手の時間という観点では非常にコストが高く、結果が何らかの意味で最適である保証もない。Section 11.8 では、より自動化された方法で高品質なネットワークを得るためのいくつかの戦略を議論する。特に、RegNetX/Y モデルにつながった ネットワーク設計空間の概念を概観する (Radosavovic *et al.*, 2020)。

11.6.7 演習

1. Fig. 11.4.1 の Inception ブロックと残差ブロックの主な違いは何か。計算量、精度、表現できる関数クラスの観点でどう比較できるか。
2. ResNet 論文 (He *et al.*, 2016) の表 1 を参照して、ネットワークのさまざまな変種を実装せよ。
3. より深いネットワークでは、ResNet はモデルの複雑さを下げるために「ボトルネック」アーキテクチャを導入している。これを実装してみよ。
4. ResNet の後続版では、著者らは「畳み込み、バッチ正規化、活性化」の構造を「バッチ正規化、活性化、畳み込み」の構造に変更した。この改良を自分で実装せよ。詳細は He *et al.* (2016) の図 1 を参照せよ。

5. 関数クラスが入れ子になっていても、なぜ関数の複雑さを無制限に増やせないのか。

11.7 密に接続されたネットワーク（DenseNet）

ResNet は、深いネットワークにおける関数をどのようにパラメータ化するかという見方を大きく変えた。*DenseNet*（dense convolutional network）は、ある意味でその論理的な拡張である (Huang *et al.*, 2017)。DenseNet の特徴は、各層がそれ以前のすべての層と接続される接続パターンと、ResNet の加算演算子ではなく連結演算を用いて、以前の層からの特徴を保持し再利用する点にある。これをどのように導くかを理解するために、少し数学に寄り道しよう。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from jax import numpy as jnp
import jax
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

11.7.1 ResNet から DenseNet へ

関数のテイラー展開を思い出しよ。点 $x = 0$ においては、次のように書ける。

$$f(x) = f(0) + x \cdot \left[f'(0) + x \cdot \left[\frac{f''(0)}{2!} + x \cdot \left[\frac{f'''(0)}{3!} + \cdots \right] \right] \right]. \tag{11.7.1}$$

重要なのは、関数を次数の高い項へと分解している点である。同様に、ResNet は関数を次のように分解する。

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}). \tag{11.7.2}$$

つまり、ResNet は f を単純な線形項と、より複雑な非線形項に分解する。2 項を超える情報を（必ずしも加算せずに）取り込みたいとしたらどうだろうか？その一つの解が DenseNet である (Huang *et al.*, 2017)。

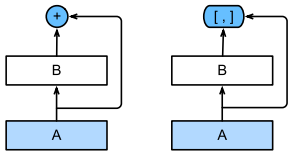


Fig. 11.7.1 ResNet（左）と DenseNet（右）の層間接続における主な違い：加算の使用と連結の使用。

Fig. 11.7.1 に示すように、ResNet と DenseNet の主な違いは、後者では出力を加算するのではなく、連結（[,] で表す）する点である。その結果、ますます複雑な関数列を適用した後の値へと、$\mathbf{x}$ を写像する。

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \ldots]. \tag{11.7.3}$$

最終的には、これらすべての関数を MLP でまとめて、特徴数を再び減らする。実装の観点では、これは非常に単純である。項を加算する代わりに、それらを連結するだけである。DenseNet という名前は、変数間の依存グラフが非常に密になることに由来する。このような連鎖の最終層は、すべての前の層と密に接続されている。密な接続は Fig. 11.7.2 に示されている。

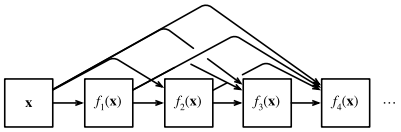


Fig. 11.7.2 DenseNet における密な接続。深くなるにつれて次元が増加することに注意。

DenseNet を構成する主な要素は、*dense block* と *transition layer* である。前者は入力と出力をどのように連結するかを定義し、後者はチャネル数を制御して大きくなりすぎないようにする。というのも、$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), \ldots]$ という拡張は、かなり高次元になりうるからである。

11.7.2 Dense Block

DenseNet は、ResNet の「バッチ正規化、活性化、畳み込み」を組み合わせた修正版の構造を使う（Section 11.6 の演習を参照）。まず、この畳み込みブロック構造を実装する。

```
def conv_block(num_channels):
    return nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=3, padding=1))
```

```
def conv_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(),
```

(continues on next page)

(continued from previous page)

```
            nn.Activation('relu'),
            nn.Conv2D(num_channels, kernel_size=3, padding=1))
    return blk
```

```
class ConvBlock(nn.Module):
    num_channels: int
    training: bool = True

    @nn.compact
    def __call__(self, X):
        Y = nn.relu(nn.BatchNorm(not self.training)(X))
        Y = nn.Conv(self.num_channels, kernel_size=(3, 3), padding=(1, 1))(Y)
        Y = jnp.concatenate((X, Y), axis=-1)
        return Y
```

```
class ConvBlock(tf.keras.layers.Layer):
    def __init__(self, num_channels):
        super(ConvBlock, self).__init__()
        self.bn = tf.keras.layers.BatchNormalization()
        self.relu = tf.keras.layers.ReLU()
        self.conv = tf.keras.layers.Conv2D(
            filters=num_channels, kernel_size=(3, 3), padding='same')

        self.listLayers = [self.bn, self.relu, self.conv]

    def call(self, x):
        y = x
        for layer in self.listLayers.layers:
            y = layer(y)
        y = tf.keras.layers.concatenate([x,y], axis=-1)
        return y
```

dense block は複数の畳み込みブロックからなり、それぞれが同じ数の出力チャネルを使う。ただし順伝播では、各畳み込みブロックの入力と出力をチャネル次元で連結する。遅延評価により、次元を自動的に調整できる。

```
class DenseBlock(nn.Module):
    def __init__(self, num_convs, num_channels):
        super(DenseBlock, self).__init__()
        layer = []
        for i in range(num_convs):
            layer.append(conv_block(num_channels))
        self.net = nn.Sequential(*layer)

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # Concatenate input and output of each block along the channels
            X = torch.cat((X, Y), dim=1)
        return X
```

```
class DenseBlock(nn.Block):
    def __init__(self, num_convs, num_channels):
        super().__init__()
        self.net = nn.Sequential()
        for _ in range(num_convs):
            self.net.add(conv_block(num_channels))

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # Concatenate input and output of each block along the channels
            X = np.concatenate((X, Y), axis=1)
        return X
```

```
class DenseBlock(nn.Module):
    num_convs: int
    num_channels: int
    training: bool = True

    def setup(self):
        layer = []
        for i in range(self.num_convs):
            layer.append(ConvBlock(self.num_channels, self.training))
        self.net = nn.Sequential(layer)

    def __call__(self, X):
        return self.net(X)
```

```
class DenseBlock(tf.keras.layers.Layer):
    def __init__(self, num_convs, num_channels):
        super(DenseBlock, self).__init__()
        self.listLayers = []
        for _ in range(num_convs):
            self.listLayers.append(ConvBlock(num_channels))

    def call(self, x):
        for layer in self.listLayers.layers:
            x = layer(x)
        return x
```

次の例では、出力チャネル数が 10 の畳み込みブロックを 2 つ持つ DenseBlock インスタンスを定義する。3 チャネルの入力を使うと、出力は $3+10+10=23$ チャネルになる。畳み込みブロックのチャネル数は、入力チャネル数に対する出力チャネル数の増加量を制御する。これは *growth rate* とも呼ばれる。

```
blk = DenseBlock(2, 10)
if tab.selected('mxnet'):
    X = np.random.uniform(size=(4, 3, 8, 8))
    blk.initialize()
if tab.selected('pytorch'):
    X = torch.randn(4, 3, 8, 8)
if tab.selected('tensorflow'):
```

(continues on next page)

(continued from previous page)

```
    X = tf.random.uniform((4, 8, 8, 3))
Y = blk(X)
Y.shape
```

```
torch.Size([4, 23, 8, 8])
```

```
blk = DenseBlock(2, 10)
if tab.selected('mxnet'):
    X = np.random.uniform(size=(4, 3, 8, 8))
    blk.initialize()
if tab.selected('pytorch'):
    X = torch.randn(4, 3, 8, 8)
if tab.selected('tensorflow'):
    X = tf.random.uniform((4, 8, 8, 3))
Y = blk(X)
Y.shape
```

```
[07:52:53] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(4, 23, 8, 8)
```

```
blk = DenseBlock(2, 10)
X = jnp.zeros((4, 8, 8, 3))
Y = blk.init_with_output(d2l.get_key(), X)[0]
Y.shape
```

```
(4, 8, 8, 23)
```

```
blk = DenseBlock(2, 10)
if tab.selected('mxnet'):
    X = np.random.uniform(size=(4, 3, 8, 8))
    blk.initialize()
if tab.selected('pytorch'):
    X = torch.randn(4, 3, 8, 8)
if tab.selected('tensorflow'):
    X = tf.random.uniform((4, 8, 8, 3))
Y = blk(X)
Y.shape
```

```
TensorShape([4, 8, 8, 23])
```

11.7.3 Transition Layer

各 dense block はチャネル数を増やすため、それをあまり多く重ねると、モデルが過度に複雑になる。*transition layer* はモデルの複雑さを制御するために使われる。これは 1×1 畳み込みを用いてチャネル数を減らする。さらに、ストライド 2 の平均プーリングによって高さと幅を半分にする。

```
def transition_block(num_channels):
    return nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=1),
        nn.AvgPool2d(kernel_size=2, stride=2))
```

```
def transition_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(), nn.Activation('relu'),
            nn.Conv2D(num_channels, kernel_size=1),
            nn.AvgPool2D(pool_size=2, strides=2))
    return blk
```

```
class TransitionBlock(nn.Module):
    num_channels: int
    training: bool = True

    @nn.compact
    def __call__(self, X):
        X = nn.BatchNorm(not self.training)(X)
        X = nn.relu(X)
        X = nn.Conv(self.num_channels, kernel_size=(1, 1))(X)
        X = nn.avg_pool(X, window_shape=(2, 2), strides=(2, 2))
        return X
```

```
class TransitionBlock(tf.keras.layers.Layer):
    def __init__(self, num_channels, **kwargs):
        super(TransitionBlock, self).__init__(**kwargs)
        self.batch_norm = tf.keras.layers.BatchNormalization()
        self.relu = tf.keras.layers.ReLU()
        self.conv = tf.keras.layers.Conv2D(num_channels, kernel_size=1)
        self.avg_pool = tf.keras.layers.AvgPool2D(pool_size=2, strides=2)

    def call(self, x):
        x = self.batch_norm(x)
        x = self.relu(x)
        x = self.conv(x)
        return self.avg_pool(x)
```

前の例の dense block の出力に対して、10 チャネルの transition layer を適用する。これにより出力チャネル数は 10 に減り、高さと幅は半分になる。

```
blk = transition_block(10)
blk(Y).shape
```

```
torch.Size([4, 10, 4, 4])
```

```
blk = transition_block(10)
blk.initialize()
blk(Y).shape
```

```
(4, 10, 4, 4)
```

```
blk = TransitionBlock(10)
blk.init_with_output(d2l.get_key(), Y)[0].shape
```

```
(4, 4, 4, 10)
```

```
blk = TransitionBlock(10)
blk(Y).shape
```

```
TensorShape([4, 4, 4, 10])
```

11.7.4 DenseNet モデル

次に、DenseNet モデルを構築する。DenseNet はまず、ResNet と同じ単一の畳み込み層と max-pooling 層を使う。

```
class DenseNet(d2l.Classifier):
    def b1(self):
        if tab.selected('mxnet'):
            net = nn.Sequential()
            net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
                nn.BatchNorm(), nn.Activation('relu'),
                nn.MaxPool2D(pool_size=3, strides=2, padding=1))
            return net
        if tab.selected('pytorch'):
            return nn.Sequential(
                nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                nn.LazyBatchNorm2d(), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        if tab.selected('tensorflow'):
            return tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(
                    64, kernel_size=7, strides=2, padding='same'),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.ReLU(),
                tf.keras.layers.MaxPool2D(
                    pool_size=3, strides=2, padding='same')])
```

```
class DenseNet(d2l.Classifier):
    def b1(self):
        if tab.selected('mxnet'):
            net = nn.Sequential()
            net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
                nn.BatchNorm(), nn.Activation('relu'),
                nn.MaxPool2D(pool_size=3, strides=2, padding=1))
            return net
        if tab.selected('pytorch'):
            return nn.Sequential(
                nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                nn.LazyBatchNorm2d(), nn.ReLU(),
                nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        if tab.selected('tensorflow'):
            return tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(
                    64, kernel_size=7, strides=2, padding='same'),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.ReLU(),
                tf.keras.layers.MaxPool2D(
                    pool_size=3, strides=2, padding='same')])
```

```
class DenseNet(d2l.Classifier):
    num_channels: int = 64
    growth_rate: int = 32
    arch: tuple = (4, 4, 4, 4)
    lr: float = 0.1
    num_classes: int = 10
    training: bool = True

    def setup(self):
        self.net = self.create_net()

    def b1(self):
        return nn.Sequential([
            nn.Conv(64, kernel_size=(7, 7), strides=(2, 2), padding='same'),
            nn.BatchNorm(not self.training),
            nn.relu,
            lambda x: nn.max_pool(x, window_shape=(3, 3),
                                  strides=(2, 2), padding='same')
        ])
```

```
class DenseNet(d2l.Classifier):
    def b1(self):
        if tab.selected('mxnet'):
            net = nn.Sequential()
            net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
                nn.BatchNorm(), nn.Activation('relu'),
                nn.MaxPool2D(pool_size=3, strides=2, padding=1))
            return net
        if tab.selected('pytorch'):
            return nn.Sequential(
                nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                nn.LazyBatchNorm2d(), nn.ReLU(),
```

(continues on next page)

(continued from previous page)

```
                nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        if tab.selected('tensorflow'):
            return tf.keras.models.Sequential([
                tf.keras.layers.Conv2D(
                    64, kernel_size=7, strides=2, padding='same'),
                tf.keras.layers.BatchNormalization(),
                tf.keras.layers.ReLU(),
                tf.keras.layers.MaxPool2D(
                    pool_size=3, strides=2, padding='same')])
```

その後、ResNet が残差ブロックからなる 4 つのモジュールを使うのと同様に、DenseNet は 4 つの dense block を使う。ResNet と同様に、各 dense block で使う畳み込み層の数を設定できる。ここでは、Section 11.6 の ResNet-18 モデルと一致するように 4 に設定する。さらに、dense block 内の畳み込み層のチャネル数（すなわち growth rate）を 32 に設定するので、各 dense block には 128 チャネルが追加される。

ResNet では、各モジュールの間でストライド 2 の残差ブロックによって高さと幅が減少する。ここでは、transition layer を使って高さと幅を半分にし、チャネル数も半分にする。ResNet と同様に、最後に global pooling 層と全結合層を接続して出力を生成する。

```
@d2l.add_to_class(DenseNet)
def __init__(self, num_channels=64, growth_rate=32, arch=(4, 4, 4, 4),
             lr=0.1, num_classes=10):
    super(DenseNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
        self.net.add(self.b1())
        for i, num_convs in enumerate(arch):
            self.net.add(DenseBlock(num_convs, growth_rate))
            # The number of output channels in the previous dense block
            num_channels += num_convs * growth_rate
            # A transition layer that halves the number of channels is added
            # between the dense blocks
            if i != len(arch) - 1:
                num_channels //= 2
                self.net.add(transition_block(num_channels))
        self.net.add(nn.BatchNorm(), nn.Activation('relu'),
                     nn.GlobalAvgPool2D(), nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.b1())
        for i, num_convs in enumerate(arch):
            self.net.add_module(f'dense_blk{i+1}', DenseBlock(num_convs,
                                                              growth_rate))
            # The number of output channels in the previous dense block
            num_channels += num_convs * growth_rate
            # A transition layer that halves the number of channels is added
            # between the dense blocks
            if i != len(arch) - 1:
```

(continues on next page)

(continued from previous page)

```
                num_channels //= 2
                self.net.add_module(f'tran_blk{i+1}', transition_block(
                    num_channels))
        self.net.add_module('last', nn.Sequential(
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.models.Sequential(self.b1())
        for i, num_convs in enumerate(arch):
            self.net.add(DenseBlock(num_convs, growth_rate))
            # The number of output channels in the previous dense block
            num_channels += num_convs * growth_rate
            # A transition layer that halves the number of channels is added
            # between the dense blocks
            if i != len(arch) - 1:
                num_channels //= 2
                self.net.add(TransitionBlock(num_channels))
        self.net.add(tf.keras.models.Sequential([
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.ReLU(),
            tf.keras.layers.GlobalAvgPool2D(),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(num_classes)]))
```

```
@d2l.add_to_class(DenseNet)
def __init__(self, num_channels=64, growth_rate=32, arch=(4, 4, 4, 4),
             lr=0.1, num_classes=10):
    super(DenseNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
        self.net.add(self.b1())
        for i, num_convs in enumerate(arch):
            self.net.add(DenseBlock(num_convs, growth_rate))
            # The number of output channels in the previous dense block
            num_channels += num_convs * growth_rate
            # A transition layer that halves the number of channels is added
            # between the dense blocks
            if i != len(arch) - 1:
                num_channels //= 2
                self.net.add(transition_block(num_channels))
        self.net.add(nn.BatchNorm(), nn.Activation('relu'),
                     nn.GlobalAvgPool2D(), nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.b1())
        for i, num_convs in enumerate(arch):
            self.net.add_module(f'dense_blk{i+1}', DenseBlock(num_convs,
                                                              growth_rate))
            # The number of output channels in the previous dense block
            num_channels += num_convs * growth_rate
            # A transition layer that halves the number of channels is added
```

(continues on next page)

(continued from previous page)

```
            # between the dense blocks
            if i != len(arch) - 1:
                num_channels //= 2
                self.net.add_module(f'tran_blk{i+1}', transition_block(
                    num_channels))
        self.net.add_module('last', nn.Sequential(
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.models.Sequential(self.b1())
        for i, num_convs in enumerate(arch):
            self.net.add(DenseBlock(num_convs, growth_rate))
            # The number of output channels in the previous dense block
            num_channels += num_convs * growth_rate
            # A transition layer that halves the number of channels is added
            # between the dense blocks
            if i != len(arch) - 1:
                num_channels //= 2
                self.net.add(TransitionBlock(num_channels))
        self.net.add(tf.keras.models.Sequential([
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.ReLU(),
            tf.keras.layers.GlobalAvgPool2D(),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(num_classes)]))
```

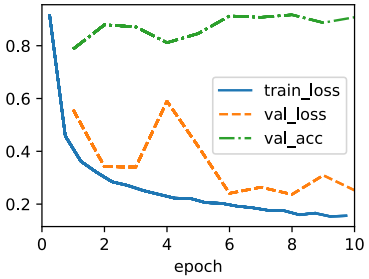
```
@d2l.add_to_class(DenseNet)
def create_net(self):
    net = self.b1()
    for i, num_convs in enumerate(self.arch):
        net.layers.extend([DenseBlock(num_convs, self.growth_rate,
                                      training=self.training)])
        # The number of output channels in the previous dense block
        num_channels = self.num_channels + (num_convs * self.growth_rate)
        # A transition layer that halves the number of channels is added
        # between the dense blocks
        if i != len(self.arch) - 1:
            num_channels //= 2
            net.layers.extend([TransitionBlock(num_channels,
                                               training=self.training)])
    net.layers.extend([
        nn.BatchNorm(not self.training),
        nn.relu,
        lambda x: nn.avg_pool(x, window_shape=x.shape[1:3],
                              strides=x.shape[1:3], padding='valid'),
        lambda x: x.reshape((x.shape[0], -1)),
        nn.Dense(self.num_classes)
    ])
    return net
```

```
@d2l.add_to_class(DenseNet)
def __init__(self, num_channels=64, growth_rate=32, arch=(4, 4, 4, 4),
             lr=0.1, num_classes=10):
    super(DenseNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
        self.net.add(self.b1())
        for i, num_convs in enumerate(arch):
            self.net.add(DenseBlock(num_convs, growth_rate))
            # The number of output channels in the previous dense block
            num_channels += num_convs * growth_rate
            # A transition layer that halves the number of channels is added
            # between the dense blocks
            if i != len(arch) - 1:
                num_channels //= 2
                self.net.add(transition_block(num_channels))
        self.net.add(nn.BatchNorm(), nn.Activation('relu'),
                     nn.GlobalAvgPool2D(), nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.b1())
        for i, num_convs in enumerate(arch):
            self.net.add_module(f'dense_blk{i+1}', DenseBlock(num_convs,
                                                              growth_rate))
            # The number of output channels in the previous dense block
            num_channels += num_convs * growth_rate
            # A transition layer that halves the number of channels is added
            # between the dense blocks
            if i != len(arch) - 1:
                num_channels //= 2
                self.net.add_module(f'tran_blk{i+1}', transition_block(
                    num_channels))
        self.net.add_module('last', nn.Sequential(
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.models.Sequential(self.b1())
        for i, num_convs in enumerate(arch):
            self.net.add(DenseBlock(num_convs, growth_rate))
            # The number of output channels in the previous dense block
            num_channels += num_convs * growth_rate
            # A transition layer that halves the number of channels is added
            # between the dense blocks
            if i != len(arch) - 1:
                num_channels //= 2
                self.net.add(TransitionBlock(num_channels))
        self.net.add(tf.keras.models.Sequential([
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.ReLU(),
            tf.keras.layers.GlobalAvgPool2D(),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(num_classes)]))
```

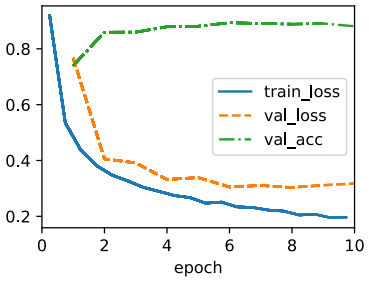
11.7.5 学習

ここではより深いネットワークを使うため、計算を簡単にする目的で、入力の高さと幅を 224 から 96 に縮小する。

```
model = DenseNet(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
trainer.fit(model, data)
```



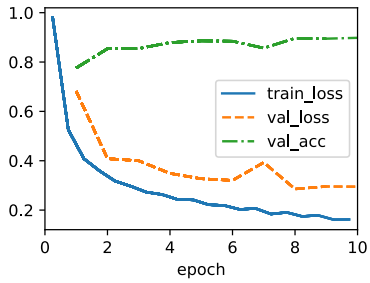

```
model = DenseNet(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
trainer.fit(model, data)
```

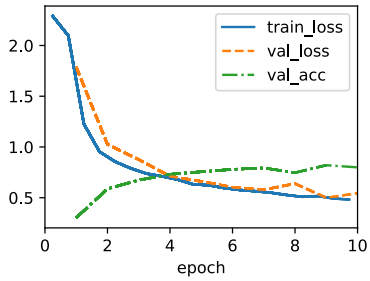




```
model = DenseNet(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
trainer.fit(model, data)
```


```
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
with d2l.try_gpu():
    model = DenseNet(lr=0.01)
    trainer.fit(model, data)
```

11.7.6 要約と考察





DenseNet を構成する主な要素は dense block と transition layer である。後者については、ネットワークを組み立てる際にチャネル数が再び縮小する transition layer を追加して、次元を制御する必要がある。層間接続の観点では、入力と出力を加算する ResNet とは対照的に、DenseNet は入力と出力をチャネル次元で連結する。これらの連結操作は、特徴を再利用して計算効率を高めるが、残念ながら GPU メモリの消費が大きくなる。その結果、DenseNet を適用するには、学習時間が増える可能性のある、よりメモリ効率の高い実装が必要になることがある (Pleiss *et al.*, 2017)。

11.7.7 演習

1. なぜ transition layer では max-pooling ではなく average pooling を使うのだろうか？
2. DenseNet の論文で挙げられている利点の一つは、モデルパラメータが ResNet より小さいことである。なぜそうなるのだろうか？
3. DenseNet が批判される問題の一つに、高いメモリ消費がある。
    1. これは本当にそうだろうか？入力形状を 224×224 に変更して、実際の GPU メモリ消費を経験的に比較してみよ。
    2. メモリ消費を減らす別の方法を考えられるか。その場合、フレームワークをどのように変更する必要があるだろうか？
4. DenseNet 論文 (Huang *et al.*, 2017) の Table 1 に示されているさまざまな DenseNet 版を実装しよ。

5. DenseNet の考え方を適用して、MLP ベースのモデルを設計しよ。それを Section 8.7 の住宅価格予測タスクに適用しよ。

11.8 畳み込みネットワークのアーキテクチャ設計

前の節では、コンピュータビジョンにおける現代的なネットワーク設計を一通り見てきた。そこで扱った研究に共通していたのは、科学者の直感に大きく依存していたことである。多くのアーキテクチャは人間の創造性に強く支えられており、深層ネットワークが提供する設計空間を体系的に探索することの比重はそれよりかなり小さいものであった。それでもなお、この ネットワーク工学的アプローチは非常に成功してきた。

AlexNet (Section 11.1) が ImageNet 上で従来のコンピュータビジョンモデルに勝って以来、同じパターンに従って設計された畳み込みブロックを積み重ねることで、非常に深いネットワークを構築することが一般的になった。特に、3×3 の畳み込みは VGG ネットワーク (Section 11.2) によって広く普及した。NiN (Section 11.3) は、局所的な非線形性を加えることで 1×1 の畳み込みでさえ有益になりうることを示した。さらに NiN は、ネットワークの最後で情報を集約する問題を、全位置にわたって集約することで解決した。GoogLeNet (Section 11.4) は、異なる畳み込み幅を持つ複数の分岐を追加し、Inception ブロックの中で VGG と NiN の利点を組み合わせた。ResNet (Section 11.6) は帰納バイアスを恒等写像（$f(x) = 0$ から）へと変えた。これにより非常に深いネットワークが可能になった。ほぼ 10 年後の今でも、ResNet の設計は依然として人気があり、その設計の優秀さを物語っている。最後に、ResNeXt (Section 11.6.5) はグループ畳み込みを追加し、パラメータ数と計算量の間でより良いトレードオフを提供した。Vision Transformer の先駆けともいえる Squeeze-and-Excitation Networks（SENets）は、位置間で効率的に情報を伝達できるようにする (Hu *et al.*, 2018)。これは、チャネルごとのグローバルな注意関数を計算することで実現された。

これまでのところ、ニューラルアーキテクチャ探索（NAS）によって得られるネットワークは扱ってこなかった (Liu *et al.*, 2018, Zoph and Le, 2016)。その理由は、通常そのコストが非常に大きく、総当たり探索、遺伝的アルゴリズム、強化学習、あるいは他の何らかのハイパーパラメータ最適化に依存するからである。固定された探索空間が与えられたとき、NAS は探索戦略を用いて、返された性能推定値に基づいてアーキテクチャを自動的に選択する。NAS の結果は単一のネットワーク実体である。EfficientNet はこの探索の注目すべき成果である (Tan and Le, 2019)。

以下では、単一の最良ネットワークを探すこととはかなり異なる考え方を議論する。これは計算コストが比較的低く、その過程で科学的な洞察も得られ、しかも結果の品質という点でもかなり有効である。では、Radosavovic *et al.* (2020) による ネットワーク設計空間を設計する戦略を見ていこう。この戦略は手作業による設計と NAS の強みを組み合わせたものである。これは ネットワークの

分布を対象にし、その分布を最適化してネットワークのファミリー全体に対して良い性能を得ることで実現される。その成果が *RegNet*、具体的には RegNetX と RegNetY であり、さらに高性能な CNN を設計するための一連の指針である。

```
from d2l import torch as d2l
import torch
from torch import nn
from torch.nn import functional as F
```

```
from d2l import mxnet as d2l
from mxnet import np, npx, init
from mxnet.gluon import nn

npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
```

```
import tensorflow as tf
from d2l import tensorflow as d2l
```

11.8.1 AnyNet の設計空間

以下の説明は、Radosavovic *et al.* (2020) の考え方にほぼ沿っているが、本書の範囲に収めるためにいくつか省略している。まず、探索対象となるネットワーク族のテンプレートが必要である。この章で扱う設計に共通する点の一つは、ネットワークが *stem*、*body*、*head* から構成されることである。stem は初期の画像処理を行い、しばしばより大きな窓サイズの畳み込みを通して処理する。body は複数のブロックからなり、生の画像から物体表現へ移るために必要な変換の大部分を担う。最後に head は、たとえば多クラス分類のための softmax 回帰器のように、これを所望の出力へ変換する。body 自体は、解像度を下げながら画像に作用する複数の stage から構成される。実際、stem もその後の各 stage も空間解像度を 4 分の 1 にする。最後に、各 stage は 1 個以上のブロックからなる。このパターンは VGG から ResNeXt まで、すべてのネットワークに共通である。実際、一般的な AnyNet ネットワークを設計するために、Radosavovic *et al.* (2020) は Fig. 11.6.5 の ResNeXt ブロックを用いた。

Fig. 11.8.1 に示した構造を詳しく見ていこう。述べたように、AnyNet は stem、body、head から成る。stem は RGB 画像（3 チャネル）を入力とし、stride が 2 の 3×3 畳み込みの後に batch norm を適用して、解像度を $r \times r$ から $r/2 \times r/2$ に半減する。さらに、body への入力となる c_0 チャネルを生成する。

このネットワークは $224 \times 224 \times 3$ 形状の ImageNet 画像でうまく動作するように設計されているため、body は 4 つの stage を通してこれを $7 \times 7 \times c_4$ に縮小する（$224/2^{1+4} = 7$ を思い出そう）。各 stage は最終的に stride 2 を持つ。最後に、

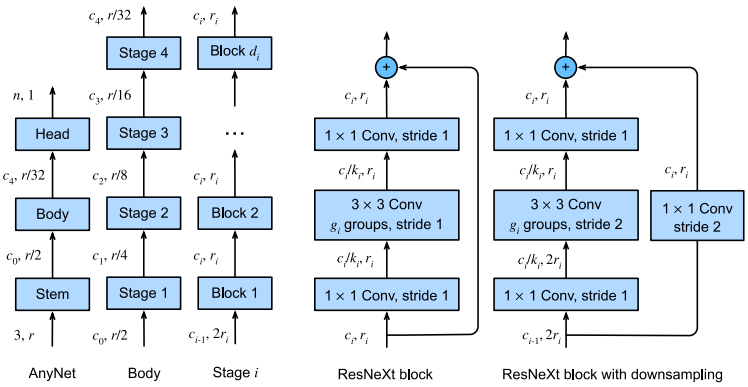


Fig. 11.8.1 AnyNet の設計空間。各矢印に沿った数値 (c, r) は、その時点でのチャネル数 c と画像の解像度 $r \times r$ を示す。左から右へ：stem、body、head からなる一般的なネットワーク構造；4 つの stage からなる body；stage の詳細構造；ブロックの 2 つの代替構造、1 つはダウンサンプリングなし、もう 1 つは各次元の解像度を半分にする。設計上の選択には、任意の stage i に対する深さ d_i、出力チャネル数 c_i、グループ数 g_i、ボトルネック比 k_i が含まれる。

head は NiN (Section 11.3) と同様に global average pooling を用い、その後に全結合層を置くという完全に標準的な設計で、n クラス分類のための n 次元ベクトルを出力する。

関連する設計上の決定の大部分はネットワークの body に内在している。body は stage ごとに進み、各 stage は Section 11.6.5 で議論したのと同じ種類の ResNeXt ブロックから構成される。ここでの設計もまた完全に一般的である。まず、stride を 2 にして解像度を半分にするブロックから始める（Fig. 11.8.1 の最右端）。これに合わせるため、ResNeXt ブロックの残差分岐は 1×1 畳み込みを通る必要がある。このブロックの後には、解像度とチャネル数の両方を変えない追加の ResNeXt ブロックが可変個続く。畳み込みブロックの設計では、わずかなボトルネックを加えるのが一般的な慣行であることに注意しよう。そのため、ボトルネック比 $k_i \geq 1$ を用いて、stage i の各ブロック内に c_i/k_i 個のチャネルを割り当てる（実験が示すように、これは実際にはあまり有効ではなく、使わないほうがよい）。最後に、ResNeXt ブロックを扱っているので、stage i におけるグループ畳み込みのグループ数 g_i も選ぶ必要がある。

この一見一般的な設計空間にも、なお多くのパラメータがある。すなわち、ブロック幅（チャネル数）$c_0, \ldots c_4$、各 stage の深さ（ブロック数）$d_1, \ldots d_4$、ボトルネック比 $k_1, \ldots k_4$、そしてグループ幅（グループ数）$g_1, \ldots g_4$ を設定できる。合計すると 17 個のパラメータになり、探索すべき構成の数は非常に大きくなる。この巨大な設計空間を効果的に縮小するための道具が必要である。ここで設計空間という概念の美しさが生きてくる。その前に、まず一般的な設計を実装しよう。

```
class AnyNet(d2l.Classifier):
    def stem(self, num_channels):
        return nn.Sequential(
            nn.LazyConv2d(num_channels, kernel_size=3, stride=2, padding=1),
            nn.LazyBatchNorm2d(), nn.ReLU())
```

```
class AnyNet(d2l.Classifier):
    def stem(self, num_channels):
        net = nn.Sequential()
        net.add(nn.Conv2D(num_channels, kernel_size=3, padding=1, strides=2),
                nn.BatchNorm(), nn.Activation('relu'))
        return net
```

```
class AnyNet(d2l.Classifier):
    arch: tuple
    stem_channels: int
    lr: float = 0.1
    num_classes: int = 10
    training: bool = True

    def setup(self):
        self.net = self.create_net()

    def stem(self, num_channels):
        return nn.Sequential([
            nn.Conv(num_channels, kernel_size=(3, 3), strides=(2, 2),
                    padding=(1, 1)),
            nn.BatchNorm(not self.training),
            nn.relu
        ])
```

```
class AnyNet(d2l.Classifier):
    def stem(self, num_channels):
        return tf.keras.models.Sequential([
            tf.keras.layers.Conv2D(num_channels, kernel_size=3, strides=2,
                                   padding='same'),
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.Activation('relu')])
```

各 stage は depth 個の ResNeXt ブロックからなり、num_channels はブロック幅を指定する。最初のブロックが入力画像の高さと幅を半分にすることに注意しよう。

```
@d2l.add_to_class(AnyNet)
def stage(self, depth, num_channels, groups, bot_mul):
    blk = []
    for i in range(depth):
        if i == 0:
            blk.append(d2l.ResNeXtBlock(num_channels, groups, bot_mul,
                use_1x1conv=True, strides=2))
        else:
```

(continues on next page)

(continued from previous page)

```
            blk.append(d2l.ResNeXtBlock(num_channels, groups, bot_mul))
    return nn.Sequential(*blk)
```

```
@d2l.add_to_class(AnyNet)
def stage(self, depth, num_channels, groups, bot_mul):
    net = nn.Sequential()
    for i in range(depth):
        if i == 0:
            net.add(d2l.ResNeXtBlock(
                num_channels, groups, bot_mul, use_1x1conv=True, strides=2))
        else:
            net.add(d2l.ResNeXtBlock(
                num_channels, num_channels, groups, bot_mul))
    return net
```

```
@d2l.add_to_class(AnyNet)
def stage(self, depth, num_channels, groups, bot_mul):
    blk = []
    for i in range(depth):
        if i == 0:
            blk.append(d2l.ResNeXtBlock(num_channels, groups, bot_mul,
                use_1x1conv=True, strides=(2, 2), training=self.training))
        else:
            blk.append(d2l.ResNeXtBlock(num_channels, groups, bot_mul,
                                        training=self.training))
    return nn.Sequential(blk)
```

```
@d2l.add_to_class(AnyNet)
def stage(self, depth, num_channels, groups, bot_mul):
    net = tf.keras.models.Sequential()
    for i in range(depth):
        if i == 0:
            net.add(d2l.ResNeXtBlock(num_channels, groups, bot_mul,
                use_1x1conv=True, strides=2))
        else:
            net.add(d2l.ResNeXtBlock(num_channels, groups, bot_mul))
    return net
```

ネットワークの stem、body、head を組み合わせることで、AnyNet の実装が完成する。

```
@d2l.add_to_class(AnyNet)
def __init__(self, arch, stem_channels, lr=0.1, num_classes=10):
    super(AnyNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
        self.net.add(self.stem(stem_channels))
        for i, s in enumerate(arch):
            self.net.add(self.stage(*s))
```

(continues on next page)

(continued from previous page)

```
        self.net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.stem(stem_channels))
        for i, s in enumerate(arch):
            self.net.add_module(f'stage{i+1}', self.stage(*s))
        self.net.add_module('head', nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.models.Sequential(self.stem(stem_channels))
        for i, s in enumerate(arch):
            self.net.add(self.stage(*s))
        self.net.add(tf.keras.models.Sequential([
            tf.keras.layers.GlobalAvgPool2D(),
            tf.keras.layers.Dense(units=num_classes)]))
```

```
@d2l.add_to_class(AnyNet)
def __init__(self, arch, stem_channels, lr=0.1, num_classes=10):
    super(AnyNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
        self.net.add(self.stem(stem_channels))
        for i, s in enumerate(arch):
            self.net.add(self.stage(*s))
        self.net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.stem(stem_channels))
        for i, s in enumerate(arch):
            self.net.add_module(f'stage{i+1}', self.stage(*s))
        self.net.add_module('head', nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.models.Sequential(self.stem(stem_channels))
        for i, s in enumerate(arch):
            self.net.add(self.stage(*s))
        self.net.add(tf.keras.models.Sequential([
            tf.keras.layers.GlobalAvgPool2D(),
            tf.keras.layers.Dense(units=num_classes)]))
```

```
@d2l.add_to_class(AnyNet)
def create_net(self):
    net = nn.Sequential([self.stem(self.stem_channels)])
    for i, s in enumerate(self.arch):
        net.layers.extend([self.stage(*s)])
    net.layers.extend([nn.Sequential([
        lambda x: nn.avg_pool(x, window_shape=x.shape[1:3],
                              strides=x.shape[1:3], padding='valid'),
```

(continues on next page)

(continued from previous page)

```
        lambda x: x.reshape((x.shape[0], -1)),
        nn.Dense(self.num_classes)])])
    return net
```

```
@d2l.add_to_class(AnyNet)
def __init__(self, arch, stem_channels, lr=0.1, num_classes=10):
    super(AnyNet, self).__init__()
    self.save_hyperparameters()
    if tab.selected('mxnet'):
        self.net = nn.Sequential()
        self.net.add(self.stem(stem_channels))
        for i, s in enumerate(arch):
            self.net.add(self.stage(*s))
        self.net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
        self.net.initialize(init.Xavier())
    if tab.selected('pytorch'):
        self.net = nn.Sequential(self.stem(stem_channels))
        for i, s in enumerate(arch):
            self.net.add_module(f'stage{i+1}', self.stage(*s))
        self.net.add_module('head', nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)
    if tab.selected('tensorflow'):
        self.net = tf.keras.models.Sequential(self.stem(stem_channels))
        for i, s in enumerate(arch):
            self.net.add(self.stage(*s))
        self.net.add(tf.keras.models.Sequential([
            tf.keras.layers.GlobalAvgPool2D(),
            tf.keras.layers.Dense(units=num_classes)]))
```

11.8.2 設計空間の分布とパラメータ

Section 11.8.1 で述べたように、設計空間のパラメータは、その設計空間に属するネットワークのハイパーパラメータである。AnyNet の設計空間で良いパラメータを見つける問題を考えてみよう。与えられた計算量（たとえば FLOPs や計算時間）に対して、単一の最良のパラメータ選択を見つけようとすることはできる。各パラメータに対して仮にたった 2 つの候補しか許さないとしても、最良解を見つけるには $2^{17} = 131072$ 通りの組合せを調べなければならない。これは明らかに、あまりにも高コストで実行不可能である。さらに悪いことに、この作業からは、ネットワークをどう設計すべきかという観点ではほとんど何も学べない。次に X-stage や shift 演算などを追加するときには、また最初からやり直しになる。しかも、学習の確率性（丸め、シャッフル、ビット誤りなど）のため、2 回の実行がまったく同じ結果を生む可能性は低いである。より良い戦略は、パラメータの選択がどのように関係すべきかについて一般的な指針を見つけることである。たとえば、ボトルネック比、チャネル数、ブロック数、グループ数、あるいは層間でのそれらの変化は、簡単な規則の集合によって支配されるのが理想である。Radosavovic *et al.* (2019) のアプローチは、次の 4 つの仮定に基づいている。

1. 一般的な設計原理は実際に存在し、それらを満たす多くのネットワークが良い性能を示すと仮定する。したがって、ネットワークの 分布を特定することは妥当な戦略である。言い換えると、干し草の山の中に良い針がたくさんあると仮定する。

2. ネットワークが良いかどうかを評価する前に、収束まで学習させる必要はない。最終的な精度の信頼できる指針として、中間結果を使えば十分である。目的関数を最適化するために（近似的な）代理指標を用いることを multi-fidelity optimization と呼ぶ (Forrester *et al.*, 2007)。したがって、設計の最適化は、データセットを数回通しただけで得られる精度に基づいて行われ、コストを大幅に削減できる。

3. 小規模で得られた結果（より小さなネットワーク）は、より大きなネットワークにも一般化する。したがって、最適化は構造的には似ているが、ブロック数やチャネル数などが少ないネットワークに対して行う。最後に、見つかったネットワークがスケールを大きくしても良い性能を示すことを確認すれば十分である。

4. 設計の各要素は近似的に因子分解できるので、その結果の品質への影響をある程度独立に推定できる。言い換えると、最適化問題は中程度に容易である。

これらの仮定により、多くのネットワークを安価に試せる。特に、構成空間から一様に サンプリングして、その性能を評価できる。その後、そうして得られたネットワークで達成可能な誤差/精度の 分布を調べることで、パラメータ選択の質を評価できる。ある設計空間から確率分布 p に従って引いたネットワークが犯す誤差について、その累積分布関数（CDF）を $F(e)$ とする。すなわち、

$$F(e, p) \stackrel{\text{def}}{=} P_{\text{net}\sim p}\{e(\text{net}) \leq e\}. \tag{11.8.1}$$

私たちの目標は、ほとんどのネットワークが非常に低い誤差率を持ち、かつ p の支持が簡潔であるような、ネットワーク上の分布 p を見つけることである。もちろん、これを正確に行うのは計算上不可能である。そこで、分布 p からネットワークの標本 $\mathcal{Z} \stackrel{\text{def}}{=} \{\text{net}_1, \ldots \text{net}_n\}$（それぞれの誤差を $e_1, \ldots, e_n$ とする）を取り、代わりに経験的 CDF $\hat{F}(e, \mathcal{Z})$ を用いる。

$$\hat{F}(e, \mathcal{Z}) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}(e_i \leq e). \tag{11.8.2}$$

Radosavovic *et al.* (2020) は、ネットワークのすべての stage に対してボトルネック比を共有する $k_i = k$ を試した。これにより、ボトルネック比を支配する 4 つのパラメータのうち 3 つが不要になる。これが性能に悪影響を与えるかどうかを評価するには、制約付き分布と非制約分布からネットワークをサンプリングし、対応する CDF を比較すればよい。すると、この制約はネットワーク分布の精度にまったく影響しないことがわかる。これは Fig. 11.8.2 の最初のパネルに示されている。同様に、ネットワークのさまざまな stage に現れるグループ幅を同じ $g_i = g$ にすることもできる。これもまた性能に影響しない。これは Fig. 11.8.2

の 2 番目のパネルに示されている。この 2 つを合わせると、自由パラメータの数は 6 つ減る。

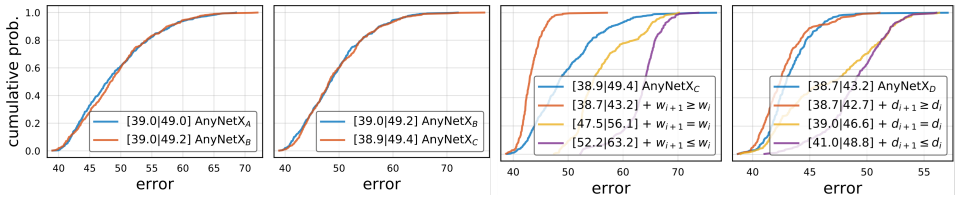


Fig. 11.8.2 設計空間の経験的誤差分布関数の比較。AnyNet$_A$ は元の設計空間、AnyNet$_B$ はボトルネック比を共有し、AnyNet$_C$ はさらにグループ幅も共有し、AnyNet$_D$ は stage 間でネットワーク深さを増加させる。左から右へ：(i) ボトルネック比を共有しても性能に影響はない；(ii) グループ幅を共有しても性能に影響はない；(iii) stage 間でネットワーク幅（チャネル数）を増やすと性能が向上する；(iv) stage 間でネットワーク深さを増やすと性能が向上する。図は Radosavovic et al. (2020) より。

次に、stage の幅と深さに関する多数の候補を減らす方法を考える。より深くなるにつれてチャネル数は増えるべき、すなわち $c_i \geq c_{i-1}$（Fig. 11.8.2 における彼らの記法では $w_{i+1} \geq w_i$）と仮定するのは妥当であり、これにより AnyNetX$_D$ が得られる。同様に、stage が進むにつれて深くなる、すなわち $d_i \geq d_{i-1}$ と仮定するのも同様に妥当であり、これにより AnyNetX$_E$ が得られる。これはそれぞれ Fig. 11.8.2 の 3 番目と 4 番目のパネルで実験的に確認できる。

11.8.3 RegNet

こうして得られる AnyNetX$_E$ の設計空間は、理解しやすい設計原理に従う単純なネットワークから成る。

- すべての stage i でボトルネック比を共有する $k_i = k$；
- すべての stage i でグループ幅を共有する $g_i = g$；
- stage 間でネットワーク幅を増やす：$c_i \leq c_{i+1}$；
- stage 間でネットワーク深さを増やす：$d_i \leq d_{i+1}$。

これで、最終的な選択肢が残る。すなわち、最終的な AnyNetX$_E$ 設計空間における上記パラメータの具体的な値をどう選ぶかである。AnyNetX$_E$ の分布から最良性能を示すネットワークを調べると、次のことが観察できる。ネットワークの幅は、理想的にはネットワーク全体のブロックインデックスに対して線形に増加する。すなわち、$c_j \approx c_0 + c_a j$ であり、ここで j はブロックインデックス、傾きは $c_a > 0$ である。stage ごとに異なるブロック幅しか選べないことを考えると、この依存関係に合わせて設計された区分的定数関数に行き着く。さらに、実験はボトルネック比 $k = 1$ が最も良いことも示しており、つまりボトルネックをまったく使わないのが望ましいということである。

異なる計算量に対する特定のネットワーク設計の詳細については、Radosavovic

et al. (2020) を参照することをお勧めする。たとえば、効果的な 32 層の RegNetX 変種は、$k = 1$（ボトルネックなし）、$g = 16$（グループ幅 16）、第 1 stage と第 2 stage のチャネル数がそれぞれ $c_1 = 32$ と $c_2 = 80$ で、深さがそれぞれ $d_1 = 4$ ブロック、$d_2 = 6$ ブロックとなるように選ばれる。この設計から得られる驚くべき洞察は、より大規模なネットワークを調べる場合でもなお成り立つということである。さらに良いことに、グローバルなチャネル活性化を持つ Squeeze-and-Excitation（SE）ネットワーク設計（RegNetY）にも当てはまる (Hu *et al.*, 2018)。

```
class RegNetX32(AnyNet):
    def __init__(self, lr=0.1, num_classes=10):
        stem_channels, groups, bot_mul = 32, 16, 1
        depths, channels = (4, 6), (32, 80)
        super().__init__(
            ((depths[0], channels[0], groups, bot_mul),
             (depths[1], channels[1], groups, bot_mul)),
            stem_channels, lr, num_classes)
```

```
class RegNetX32(AnyNet):
    def __init__(self, lr=0.1, num_classes=10):
        stem_channels, groups, bot_mul = 32, 16, 1
        depths, channels = (4, 6), (32, 80)
        super().__init__(
            ((depths[0], channels[0], groups, bot_mul),
             (depths[1], channels[1], groups, bot_mul)),
            stem_channels, lr, num_classes)
```

```
class RegNetX32(AnyNet):
    lr: float = 0.1
    num_classes: int = 10
    stem_channels: int = 32
    arch: tuple = ((4, 32, 16, 1), (6, 80, 16, 1))
```

```
class RegNetX32(AnyNet):
    def __init__(self, lr=0.1, num_classes=10):
        stem_channels, groups, bot_mul = 32, 16, 1
        depths, channels = (4, 6), (32, 80)
        super().__init__(
            ((depths[0], channels[0], groups, bot_mul),
             (depths[1], channels[1], groups, bot_mul)),
            stem_channels, lr, num_classes)
```

各 RegNetX stage が、解像度を段階的に下げつつ出力チャネル数を増やしていくことがわかる。

```
RegNetX32().layer_summary((1, 1, 96, 96))
```

```
Sequential output shape:     torch.Size([1, 32, 48, 48])
```

(continues on next page)

(continued from previous page)

```
Sequential output shape:     torch.Size([1, 32, 24, 24])
Sequential output shape:     torch.Size([1, 80, 12, 12])
Sequential output shape:     torch.Size([1, 10])
```

```
RegNetX32().layer_summary((1, 1, 96, 96))
```

```
[07:49:42] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
Sequential output shape:     (1, 32, 48, 48)
Sequential output shape:     (1, 32, 24, 24)
Sequential output shape:     (1, 80, 12, 12)
GlobalAvgPool2D output shape:        (1, 80, 1, 1)
Dense output shape:  (1, 10)
```

```
RegNetX32(training=False).layer_summary((1, 96, 96, 1))
```

```
Sequential output shape:     (1, 48, 48, 32)
Sequential output shape:     (1, 24, 24, 32)
Sequential output shape:     (1, 12, 12, 80)
Sequential output shape:     (1, 10)
```

```
RegNetX32().layer_summary((1, 96, 96, 1))
```

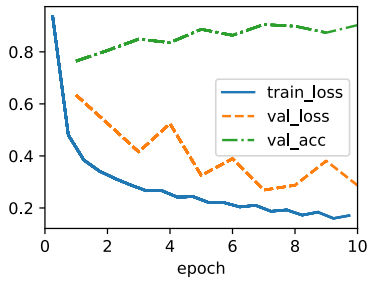
```
Sequential output shape:     (1, 48, 48, 32)
Sequential output shape:     (1, 24, 24, 32)
Sequential output shape:     (1, 12, 12, 80)
Sequential output shape:     (1, 10)
```

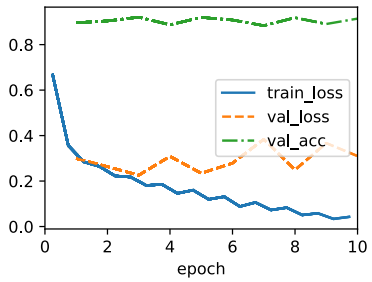
11.8.4 学習

Fashion-MNIST データセットで 32 層の RegNetX を学習するのは、これまでと同じである。

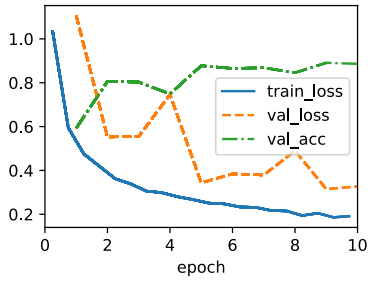
```
model = RegNetX32(lr=0.05)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
trainer.fit(model, data)
```

```
model = RegNetX32(lr=0.05)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
trainer.fit(model, data)
```





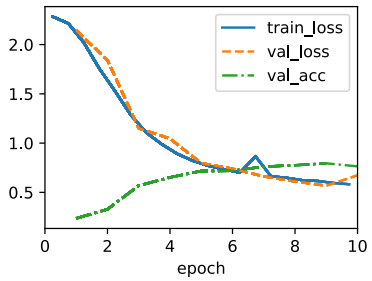

```
model = RegNetX32(lr=0.05)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
trainer.fit(model, data)
```




```
trainer = d2l.Trainer(max_epochs=10)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
with d2l.try_gpu():
    model = RegNetX32(lr=0.01)
    trainer.fit(model, data)
```

11.8.5 議論

局所性や平行移動不変性（Section 10.1）のような、視覚に対する望ましい帰納バイアス（仮定や好み）を持つため、CNN はこの分野で支配的なアーキテクチャでした。これは LeNet から、Transformers (Section 14.7) (Dosovitskiy *et al.*, 2021,



Touvron *et al.*, 2021) が精度の面で CNN を上回り始めるまで続きました。最近の vision Transformer の進歩の多くは CNN に 移植可能ですが (Liu *et al.*, 2022)、それはより高い計算コストを伴う場合に限られる。同様に重要なのは、NVIDIA Ampere や Hopper といった最近のハードウェア最適化が、Transformer に有利な差をさらに広げていることである。

Transformer は、CNN に比べて局所性や平行移動不変性に対する帰納バイアスの度合いがかなり低いことに注意する価値がある。学習された構造が優勢になったのは、最大 50 億枚の画像を含む LAION-400m や LAION-5B (Schuhmann *et al.*, 2022) のような大規模画像コレクションが利用可能になったことも大きな要因である。驚くべきことに、この文脈でより重要な研究の中には MLP さえ含まれている (Tolstikhin *et al.*, 2021)。

要するに、vision Transformer (Section 14.8) は現在、大規模画像分類における最先端性能の面で先行しており、スケーラビリティは帰納バイアスに勝ることを示している (Dosovitskiy *et al.*, 2021)。これには、multi-head self-attention (Section 14.5) を用いた大規模 Transformer の事前学習 (Section 14.9) も含まれる。これらの章をぜひ読み進めて、より詳細な議論に触れよ。

11.8.6 演習

1. stage の数を 4 に増やせ。より深く、より良い性能を示す RegNetX を設計できるか。
2. ResNeXt ブロックを ResNet ブロックに置き換えて、RegNet を De-ResNeXt 化せよ。新しいモデルの性能はどうなるか。
3. RegNetX の設計原理を 破ることで、「VioNet」ファミリーの複数の実体を実装せよ。それらの性能はどうなるか。(d_i, c_i, g_i, b_i) のうち、最も重要な要因はどれか。
4. あなたの目標は「完璧な」MLP を設計することである。上で導入した設計原理を使って、良いアーキテクチャを見つけられるだろうか。小規模ネットワークから大規模ネットワークへ外挿することは可能だろうか。

12 再帰ニューラルネットワーク

これまで、私たちは主として固定長データに焦点を当ててきた。Section 6 と Section 7 で線形回帰とロジスティック回帰を導入し、Section 8 で多層パーセプトロンを導入したときには、各特徴ベクトル $\mathbf{x}_i$ が $x_1, \ldots, x_d$ という固定個数の成分から成り、各数値特徴 x_j が特定の属性に対応していると仮定してよいことを前提としていた。このようなデータセットは、しばしば 表形式データと呼ばれる。なぜなら、各例 i がそれぞれ 1 つの行を持ち、各属性がそれぞれ 1 つの列を持つ表として並べることができるからである。重要なのは、表形式データでは、列の間に特定の構造をほとんど仮定しないことである。

その後、Section 10 では画像データへと進んだ。そこでは入力は画像内の各座標における生の画素値から成る。画像データは、典型的な表形式データセットの条件にほとんど当てはまらなかった。そこで、階層構造と不変性を扱うために畳み込みニューラルネットワーク（CNN）を用いる必要があった。しかし、それでもデータは固定長であった。Fashion-MNIST の各画像は 28×28 の画素値グリッドとして表現される。さらに、私たちの目的は 1 枚の画像だけを見て単一の予測を出力するモデルを作ることであった。では、動画のような画像の系列に直面したときや、画像キャプショニングのように系列構造を持つ予測を生成する必要があるときには、どうすればよいのだろうか。

非常に多くの学習タスクでは、系列データを扱う必要がある。画像キャプショニング、音声合成、音楽生成はいずれも、モデルが系列から成る出力を生成することを要求する。他の分野、たとえば時系列予測、動画解析、音楽情報検索では、モデルは系列である入力から学習しなければならない。こうした要求はしばしば同時に現れる。たとえば、ある自然言語から別の自然言語へ文章を翻訳する、対話を行う、あるいはロボットを制御するといったタスクでは、モデルが系列構造を持つデータを入力し、かつ出力することが求められる。

再帰ニューラルネットワーク（RNN）は、再帰的な接続によって系列の動態を捉える深層学習モデルである。これはネットワーク内のノードの循環と考えることができる。最初は直感に反するように思えるかもしれない。そもそも、ニューラルネットワークの順伝播的な性質こそが、計算順序を曖昧でないものにしているからである。しかし、再帰的な辺は厳密に定義されており、そのような曖昧さが生じないことが保証されている。再帰ニューラルネットワークは、時間ステップ（または系列ステップ）にわたって 展開され、各ステップで 同じ基本パラメータが適用される。標準的な接続が、各層の活性を 同じ時間ステップ内で次の層へ伝播するために 同期的に適用されるのに対し、再帰接続は動的であり、隣接する時間ステップ間で情報を受け渡す。Fig. 12.1 の展開図が示すように、RNN は、各層のパラメータ（通常の接続と再帰接続の両方）が時

間ステップ間で共有される順伝播ニューラルネットワークと考えることができる。

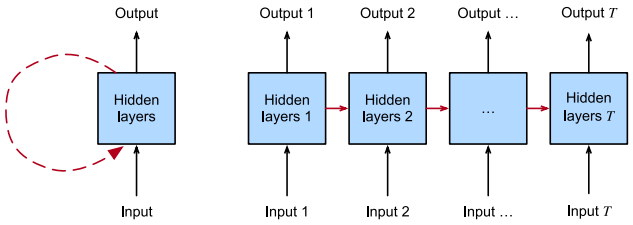


Fig. 12.1 左では再帰接続が循環辺として描かれている。右では、RNN を時間ステップに沿って展開している。ここでは、再帰辺は隣接する時間ステップにまたがり、通常の接続は同期的に計算される。

ニューラルネットワーク全般と同様に、RNN には長い学際的な歴史がある。もともとは脳のモデルとして認知科学者によって広められ、その後、機械学習コミュニティで実用的なモデリング手法として採用された。深層学習全般についてそうするように、本書では機械学習の視点を採用し、2010 年代に、手書き認識 (Graves *et al.*, 2008)、機械翻訳 (Sutskever *et al.*, 2014)、医療診断の認識 (Lipton *et al.*, 2016) といった多様なタスクでの画期的な成果によって人気を博した実用的な道具として RNN に焦点を当てる。より詳しい背景資料を求める読者には、公開されている包括的なレビュー (Lipton *et al.*, 2015) を参照することを勧める。また、系列性は RNN に固有のものではないことも指摘しておく。たとえば、すでに導入した CNN も、長さが可変のデータ、たとえば解像度が異なる画像を扱えるように適応させることができる。さらに、RNN は近年、Transformer モデルにかなりの市場シェアを譲っており、それについては Section 14 で扱う。しかし、RNN は深層学習において複雑な系列構造を扱うための標準的なモデルとして台頭し、今日に至るまで系列モデリングの定番モデルであり続けている。RNN の物語と系列モデリングの物語は切り離せないほど密接に結びついており、この章は RNN についての章であると同時に、系列モデリング問題の ABC についての章でもある。

ある重要な洞察が、系列モデリングの革命への道を開いた。機械学習における多くの基本的なタスクでは、入力と目標を固定長ベクトルとして容易に表現することはできないが、それでもしばしば、固定長ベクトルの長さ可変な系列として表現できる。たとえば、文書は単語の系列として表現できる。医療記録はしばしば、出来事の系列（受診、投薬、処置、検査、診断）として表現できる。動画は、静止画像の長さ可変な系列として表現できる。

系列モデルは数多くの応用分野に現れているが、この分野の基礎研究は主として自然言語処理の中核タスクにおける進歩によって牽引されてきた。したがって、この章全体を通して、説明と例はテキストデータに焦点を当てる。これらの例の要領をつかめば、他のデータモダリティへのモデルの適用は比較的容易なはずである。次のいくつかの節では、系列に関する基本的な記法と、系列構造を持つモデル出力の品質を評価するためのいくつかの評価指標を導入する。その後、言語モデルの基本概念を議論し、この議論を用いて最初の RNN モデ

ルを動機づける。最後に、RNN を通して逆伝播するときの勾配計算方法を説明し、このようなネットワークの学習時によく遭遇するいくつかの課題を検討して、Section 13 で続く現代的な RNN アーキテクチャの動機づけを行う。

12.1 シーケンスの扱い

これまで、入力が単一の特徴ベクトル $\mathbf{x} \in \mathbb{R}^d$ からなるモデルに焦点を当ててきた。シーケンスを処理できるモデルを開発する際の視点の主な変化は、入力が順序づけられた特徴ベクトルの列 $\mathbf{x}_1, \ldots, \mathbf{x}_T$ からなるものに注目するようになることである。ここで各特徴ベクトル $\mathbf{x}_t$ は、$\mathbb{R}^d$ に属する時刻 $t \in \mathbb{Z}^+$ によってインデックス付けされる。

データセットの中には、単一の巨大なシーケンスからなるものもある。たとえば、気候科学者が利用できるかもしれない、極めて長いセンサー観測のストリームを考えてみよう。このような場合、あらかじめ定めた長さの部分シーケンスをランダムにサンプリングすることで、学習用データセットを作成できる。より一般的には、データはシーケンスの集合として与えられる。以下の例を考えてみよう。(i) 文書の集合で、それぞれが単語のシーケンスとして表され、各文書はそれぞれ異なる長さ T_i を持つ。(ii) 入院中の患者の滞在をシーケンスとして表現する場合で、各滞在は複数のイベントからなり、シーケンス長はおおむね滞在期間に依存する。

以前、個々の入力を扱うときには、それらが同じ基礎分布 $P(X)$ から独立にサンプリングされると仮定した。シーケンス全体（たとえば文書全体や患者の軌跡全体）が独立にサンプリングされると仮定することは今でもできるが、各時刻に到着するデータ同士が互いに独立であるとは仮定できない。たとえば、文書の後半に現れそうな単語は、その文書の前半に現れた単語に強く依存する。患者が入院 10 日目に受ける可能性の高い治療は、直前の 9 日間に何が起きたかに強く依存する。

これは驚くことではない。もしシーケンスの要素が関連していないと信じるなら、そもそもそれらをシーケンスとしてモデル化しようとはしないはずである。検索ツールや現代的なメールクライアントで人気のある自動補完機能の有用性を考えてみよう。これらが有用なのは、ある初期の接頭辞が与えられたときに、シーケンスのもっともらしい続きが何かを（不完全ではあるが、ランダム推測よりは良く）予測できることが多いからである。多くのシーケンスモデルでは、独立性や定常性すら要求しない。代わりに、シーケンスそのものが、シーケンス全体にわたるある固定された基礎分布からサンプリングされることだけを要求する。

この柔軟な考え方により、たとえば次のような現象を扱える。(i) 文書の冒頭と末尾で見た目が大きく異なること。(ii) 入院期間を通じて患者の状態が回復方向または死亡方向へ変化していくこと。(iii) レコメンダシステムとの継続的なやり取りの中で、顧客の嗜好が予測可能な形で変化していくこと。

固定されたターゲット y を、順序構造を持つ入力から予測したいことがある（たとえば、映画レビューに基づく感情分類）。別のときには、順序構造を持つターゲット $(y_1, \ldots, y_T)$ を固定入力から予測したいことがある（たとえば、画像キャプション生成）。さらに別のときには、順序構造を持つ入力に基づいて順序構造を持つターゲットを予測したいこともある（たとえば、機械翻訳や動画キャプション生成）。このようなシーケンス対シーケンスのタスクには 2 つの形がある。(i) *aligned*：各時刻の入力が対応するターゲットと整列している場合（たとえば、品詞タグ付け）。(ii) *unaligned*：入力とターゲットが必ずしも 1 対 1 に対応しない場合（たとえば、機械翻訳）。

あらゆる種類のターゲットを扱うことを考える前に、最も直接的な問題に取り組もう。それは、教師なし密度モデリング（*sequence modeling* とも呼ばれる）である。ここでは、シーケンスの集合が与えられたとき、任意のシーケンスがどれくらい起こりやすいかを与える確率質量関数、すなわち $p(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ を推定することが目標である。

```
%load_ext d2lbook.tab
tab.interact_select('mxnet', 'pytorch', 'tensorflow', 'jax')
```

```
%matplotlib inline
from d2l import torch as d2l
import torch
from torch import nn
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, np, npx, gluon, init
from mxnet.gluon import nn
npx.set_np()
```

```
%matplotlib inline
from d2l import jax as d2l
import jax
from jax import numpy as jnp
import numpy as np
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf
```

12.1.1 自己回帰モデル

順序構造を持つデータを扱うために設計された特殊なニューラルネットワークを導入する前に、実際のシーケンスデータを見て、基本的な直感と統計的手法をいくつか構築してみよう。特に、FTSE 100 指数の株価データに注目する（Fig. 12.1.1）。各 時刻 $t \in \mathbb{Z}^+$ において、その時点での指数の価格 x_t を観測する。

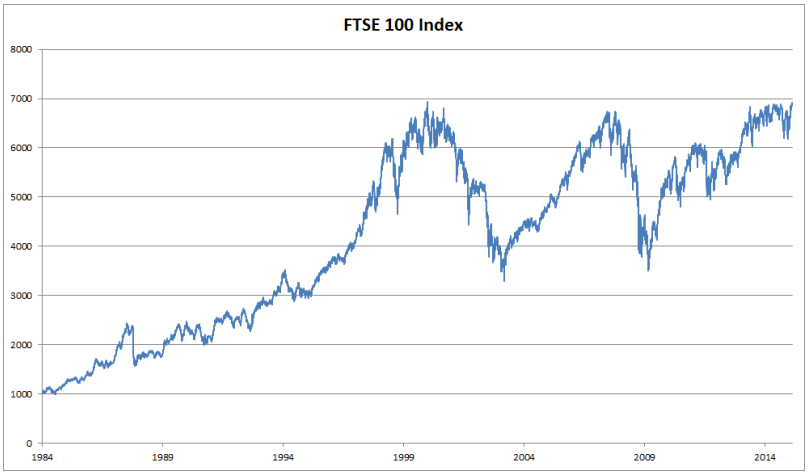


Fig. 12.1.1 約 30 年にわたる FTSE 100 指数。

ここで、あるトレーダーが短期売買を行いたいとする。つまり、次の時刻に指数が上がると考えるか下がると考えるかに応じて、戦略的に指数に入ったり出たりしたいのである。ほかに特徴量（ニュース、財務報告データなど）がない場合、次の値を予測するために利用できる唯一の信号は、これまでの価格履歴である。したがって、そのトレーダーが知りたいのは、次の時刻に指数が取りうる価格についての確率分布

$$P(x_t \mid x_{t-1}, \ldots, x_1) \tag{12.1.1}$$

である。連続値の確率変数に対する分布全体を推定するのは難しいかもしれないが、トレーダーは分布のいくつかの重要な統計量、特に期待値と分散に注目できれば満足だろう。条件付き期待値

を推定する簡単な戦略の 1 つは、線形回帰モデルを適用することである（Section 6.1 を参照）。このように、ある信号の値をその信号自身の過去の値に回帰させるモデルは、自然に 自己回帰モデルと呼ばれる。ただし大きな問題が 1 つある。入力の数 $x_{t-1}, \ldots, x_1$ は t に応じて変化するのである。言い換えると、入力の数は遭遇するデータ量に応じて増えていく。したがって、履歴データを訓練セットとして扱いたい場合、各例が異なる数の特徴量を持つという問題が残る。この章の以降の多くは、このような 自己回帰モデリング問題、すなわち関心の対象が $P(x_t \mid x_{t-1}, \ldots, x_1)$ またはこの分布の統計量である場合に、これらの課題を克服するための手法を中心に展開する。

よく繰り返し現れる戦略がいくつかある。まず、長いシーケンス $x_{t-1}, \ldots, x_1$ が利用可能であっても、近い将来を予測するときに、履歴をそんなに遠くまで遡る必要はないかもしれないと考えられる。この場合、長さ τ の窓に条件付けし、$x_{t-1}, \ldots, x_{t-\tau}$ の観測だけを使うことにする。直ちに得られる利点は、少なくとも $t > \tau$ では、引数の数が常に同じになることである。これにより、固定長ベクトルを入力として必要とする任意の線形モデルや深層ネットワークを訓練できる。第二に、過去の観測の何らかの要約 h_t を保持し（Fig. 12.1.2 を参照）、

同時に予測 $\hat{x}_t$ に加えて h_t を更新するモデルを開発することもできる。これにより、$\hat{x}_t = P(x_t \mid h_t)$ によって x_t を推定するだけでなく、$h_t = g(h_{t-1}, x_{t-1})$ のような更新も行うモデルにつながる。h_t は観測されないため、これらのモデルは 潜在自己回帰モデルとも呼ばれる。

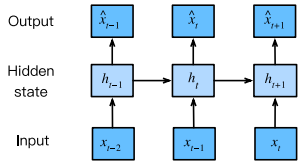


Fig. 12.1.2 潜在自己回帰モデル。

履歴データから訓練データを構成するには、通常、窓をランダムにサンプリングして例を作成する。一般に、時間が止まっているとは考えない。しかし、x_t の具体的な値は変化しても、各観測がそれ以前の観測に基づいて生成されるダイナミクス自体は変化しないと仮定することがよくある。統計学者は、このように変化しないダイナミクスを 定常と呼ぶ。

12.1.2 シーケンスモデル

ときには、特に言語を扱う場合、シーケンス全体の同時確率を推定したいことがある。これは、単語のような離散的な トークンから構成されるシーケンスを扱うときによくあるタスクである。一般に、このような推定関数は シーケンスモデルと呼ばれ、自然言語データに対しては 言語モデルと呼ばれる。シーケンスモデリングの分野は自然言語処理によって大きく牽引されてきたため、非言語データを扱う場合であっても、シーケンスモデルを「言語モデル」と呼ぶことがよくある。言語モデルはさまざまな理由で有用である。文の尤度を評価したいことがある。たとえば、機械翻訳システムや音声認識システムが生成した 2 つの候補出力の自然さを比較したいかもしれない。しかし、言語モデリングは尤度を 評価する能力だけでなく、シーケンスを サンプリングする能力、さらには最も尤もらしいシーケンスを最適化する能力も与えてくれる。

言語モデリングは一見すると自己回帰問題のようには見えないかもしれないが、確率の連鎖律を適用して、シーケンス $p(x_1, \ldots, x_T)$ の同時密度を左から右へと条件付き密度の積に分解することで、言語モデリングを自己回帰予測に還元できる。

$$P(x_1, \ldots, x_T) = P(x_1) \prod_{t=2}^{T} P(x_t \mid x_{t-1}, \ldots, x_1). \tag{12.1.2}$$

単語のような離散信号を扱う場合、自己回帰モデルは確率的分類器でなければならず、左側の文脈が与えられたときに次に来る単語について、語彙全体にわたる完全な確率分布を出力する。

マルコフモデル

ここで、上で述べた戦略、すなわちシーケンス全体の履歴 $x_{t-1}, \ldots, x_1$ ではなく、直前の τ ステップ $x_{t-1}, \ldots, x_{t-\tau}$ のみに条件付ける方法を使いたいとする。予測能力を失うことなく、直前の τ ステップより前の履歴を捨てられるとき、そのシーケンスは マルコフ条件を満たすと言う。つまり、未来は最近の履歴が与えられたとき過去から条件付き独立であるということである。$\tau = 1$ のとき、データは *1* 次マルコフモデルによって特徴づけられると言い、$\tau = k$ のとき、データは k^{th}-order Markov model によって特徴づけられると言う。1 次マルコフ条件が成り立つ（$\tau = 1$）場合、同時確率の分解は、各単語が直前の 単語に条件付けられた確率の積になる。

$$P(x_1, \ldots, x_T) = P(x_1) \prod_{t=2}^{T} P(x_t \mid x_{t-1}). \tag{12.1.3}$$

マルコフ条件が成り立つかのように進むモデルを使うことは、実際にはそれが近似的にしか真でないと分かっていても、しばしば有用である。実際のテキスト文書では、左側の文脈をより多く含めるほど情報は増え続ける。しかし、その増加は急速に逓減する。したがって、ときには、k^{th}-order Markov condition に依存するモデルを訓練することで、計算上および統計上の困難を回避するという妥協をする。今日の巨大な RNN や Transformer ベースの言語モデルでさえ、数千語を超える文脈を取り込むことはめったにない。

離散データでは、真のマルコフモデルは、各文脈において各単語が何回現れたかを数えるだけで、$P(x_t \mid x_{t-1})$ の相対頻度推定を生成する。データが（言語のように）離散値のみをとる場合、最も尤もらしい単語列は動的計画法を用いて効率的に計算できる。

デコードの順序

なぜテキストシーケンス $P(x_1, \ldots, x_T)$ の分解を、左から右への条件付き確率の連鎖として表したのか、不思議に思うかもしれない。右から左や、あるいは別の一見ランダムな順序ではだめなのだろうか。原理的には、$P(x_1, \ldots, x_T)$ を逆順に展開しても問題ない。その結果は有効な分解になる。

$$P(x_1, \ldots, x_T) = P(x_T) \prod_{t=T-1}^{1} P(x_t \mid x_{t+1}, \ldots, x_T). \tag{12.1.4}$$

しかし、言語モデリングのタスクでは、私たちが読む方向と同じ方向でテキストを分解すること（多くの言語では左から右、アラビア語やヘブライ語では右から左）が好まれる理由がいくつもある。第一に、これは単に私たちにとって考えやすい自然な方向だからである。私たちは毎日テキストを読み、その過程は次にどの単語やフレーズが来そうかを予測する能力によって導かれている。誰かの文を何度も最後まで言い切ってしまった経験を思い浮かべてみよう。したがって、このような順方向のデコードを特に好む他の理由がなかったとしても、この順序で予測するときに何が尤もらしいかについて、私たちの直感がより働くというだけでも有用である。

第二に、順序に沿って分解することで、同じ言語モデルを使って任意に長いシーケンスに確率を割り当てられる。ステップ 1 から t までの確率を $t+1$ 番目の単語まで拡張するには、単に追加トークンの条件付き確率を、以前のトークンに条件付けて掛ければよいのである：$P(x_{t+1},\ldots,x_1) = P(x_t,\ldots,x_1)\cdot P(x_{t+1} \mid x_t,\ldots,x_1)$。

第三に、任意の他の位置にある単語よりも、隣接する単語を予測するほうが強力な予測モデルを持てる。すべての分解順序は有効であるが、それらがすべて同じくらい容易な予測モデリング問題を表しているとは限らない。これは言語だけでなく、他の種類のデータにも当てはまる。たとえば、データが因果構造を持つ場合である。たとえば、未来の出来事が過去に影響を与えることはないと考える。したがって、x_t を変えると、その先の x_{t+1} で何が起こるかには影響を与えられるかもしれないが、その逆はできない。つまり、x_t を変えても、過去の出来事の分布は変わらない。ある文脈では、このため $P(x_{t+1} \mid x_t)$ を予測するほうが $P(x_t \mid x_{t+1})$ を予測するより容易になる。たとえば、場合によっては、ある加法ノイズ ϵ に対して $x_{t+1} = f(x_t) + \epsilon$ と書ける一方で、その逆は成り立たないことがある (Hoyer *et al.*, 2009)。これは非常に良い知らせである。というのも、通常、私たちが推定したいのは前向きの方向だからである。Peters *et al.* (2017) の本には、この話題についてさらに詳しい説明がある。ここではその表面をかすめる程度にしか触れない。

12.1.3 訓練

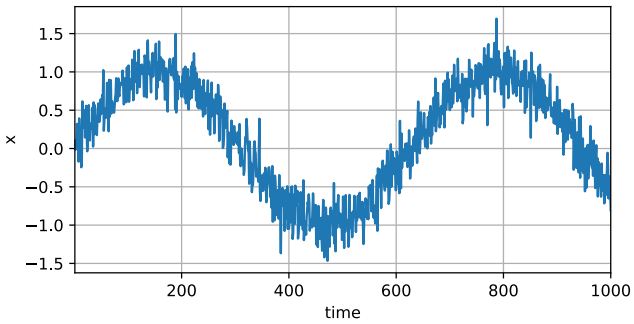
テキストデータに注目する前に、まず連続値の合成データで試してみよう。

ここでは、1000 個の合成データが、時刻に 0.01 を掛けた値に適用した三角関数の sin に従うものとする。問題を少し面白くするために、各サンプルに加法ノイズを加える。このシーケンスから、特徴量とラベルからなる訓練例を抽出する。

```
class Data(d2l.DataModule):
    def __init__(self, batch_size=16, T=1000, num_train=600, tau=4):
        self.save_hyperparameters()
        self.time = d2l.arange(1, T + 1, dtype=d2l.float32)
        if tab.selected('mxnet', 'pytorch'):
            self.x = d2l.sin(0.01 * self.time) + d2l.randn(T) * 0.2
        if tab.selected('tensorflow'):
            self.x = d2l.sin(0.01 * self.time) + d2l.normal([T]) * 0.2
        if tab.selected('jax'):
            key = d2l.get_key()
            self.x = d2l.sin(0.01 * self.time) + jax.random.normal(key,
                                                                   [T]) * 0.2
```

```
data = Data()
d2l.plot(data.time, data.x, 'time', 'x', xlim=[1, 1000], figsize=(6, 3))
```

まず、データが τ^{th}-order Markov condition を満たすかのように振る舞うモデル、

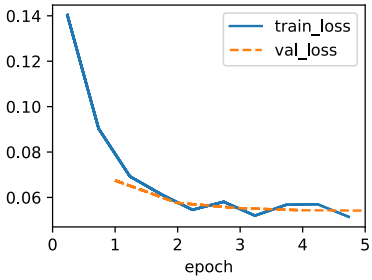


つまり過去 τ 個の観測だけを使って x_t を予測するモデルを試す。したがって、各時刻について、ラベル $y = x_t$ と特徴量 $\mathbf{x}_t = [x_{t-\tau}, \ldots, x_{t-1}]$ を持つ例が得られる。鋭い読者は、$y_1, \ldots, y_\tau$ に対して十分な履歴がないため、これにより $1000-\tau$ 個の例が得られることに気づいたかもしれない。最初の τ 個のシーケンスにゼロをパディングすることもできるが、ここでは簡単のためにそれらを捨てる。結果として得られるデータセットには $T - \tau$ 個の例が含まれ、モデルへの各入力のシーケンス長は τ である。私たちは最初の 600 個の例に対するデータイテレータを作成し、sin 関数の 1 周期分をカバーする。

```
@d2l.add_to_class(Data)
def get_dataloader(self, train):
    features = [self.x[i : self.T-self.tau+i] for i in range(self.tau)]
    self.features = d2l.stack(features, 1)
    self.labels = d2l.reshape(self.x[self.tau:], (-1, 1))
    i = slice(0, self.num_train) if train else slice(self.num_train, None)
    return self.get_tensorloader([self.features, self.labels], train, i)
```

この例では、モデルは標準的な線形回帰になる。

```
model = d2l.LinearRegression(lr=0.01)
trainer = d2l.Trainer(max_epochs=5)
trainer.fit(model, data)
```



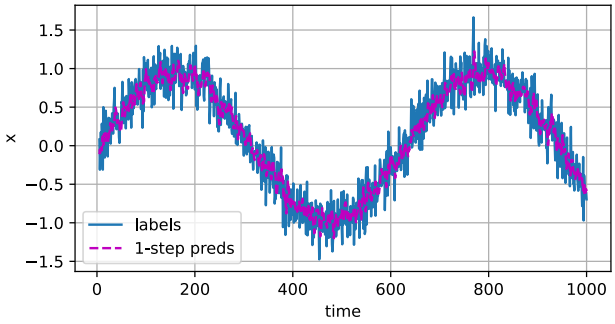
12.1.4 予測

モデルを評価するために、まず 1 ステップ先予測でどの程度うまくいくかを確認する。

```
onestep_preds = d2l.numpy(model(data.features))
d2l.plot(data.time[data.tau:], [data.labels, onestep_preds], 'time', 'x',
         legend=['labels', '1-step preds'], figsize=(6, 3))
```


```
onestep_preds = d2l.numpy(model(data.features))
d2l.plot(data.time[data.tau:], [data.labels, onestep_preds], 'time', 'x',
         legend=['labels', '1-step preds'], figsize=(6, 3))
```


```
onestep_preds = model.apply({'params': trainer.state.params}, data.features)
d2l.plot(data.time[data.tau:], [data.labels, onestep_preds], 'time', 'x',
         legend=['labels', '1-step preds'], figsize=(6, 3))
```




```
onestep_preds = d2l.numpy(model(data.features))
d2l.plot(data.time[data.tau:], [data.labels, onestep_preds], 'time', 'x',
         legend=['labels', '1-step preds'], figsize=(6, 3))
```


これらの予測は良さそうに見える。終盤の $t = 1000$ 付近でも同様である。

しかし、時刻 604（`n_train + tau`）までしかシーケンスデータを観測しておらず、その先を数ステップ予測したいとしたらどうだろうか。残念ながら、時刻 609 の 1 ステップ先予測を直接計算することはできない。対応する入力が分からないからである。x_{604} までしか見ていないためである。この問題は、以前の予測をモデルへの入力として次の予測に使い、1 ステップずつ先へ進めて、目的の時刻に到達することで解決できる。

$$\begin{aligned}
\hat{x}_{605} &= f(x_{601}, x_{602}, x_{603}, x_{604}),\\
\hat{x}_{606} &= f(x_{602}, x_{603}, x_{604}, \hat{x}_{605}),\\
\hat{x}_{607} &= f(x_{603}, x_{604}, \hat{x}_{605}, \hat{x}_{606}),\\
\hat{x}_{608} &= f(x_{604}, \hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}),\\
\hat{x}_{609} &= f(\hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}, \hat{x}_{608}),\\
&\vdots
\end{aligned} \tag{12.1.5}$$

一般に、観測されたシーケンス $x_1, \ldots, x_t$ に対して、時刻 $t+k$ における予測出力 $\hat{x}_{t+k}$ を k ステップ先予測と呼ぶ。x_{604} まで観測しているので、その k ステップ先予測は $\hat{x}_{604+k}$ である。言い換えると、多段先予測を行うには、自分自身の予測を使い続けなければならない。これがどの程度うまくいくか見てみよう。

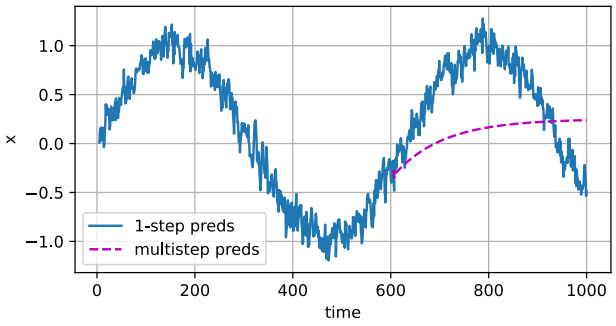
```
multistep_preds = d2l.zeros(data.T)
multistep_preds[:] = data.x
for i in range(data.num_train + data.tau, data.T):
    multistep_preds[i] = model(
        d2l.reshape(multistep_preds[i-data.tau : i], (1, -1)))
multistep_preds = d2l.numpy(multistep_preds)
```

```
multistep_preds = d2l.zeros(data.T)
multistep_preds[:] = data.x
for i in range(data.num_train + data.tau, data.T):
    multistep_preds[i] = model(
        d2l.reshape(multistep_preds[i-data.tau : i], (1, -1)))
multistep_preds = d2l.numpy(multistep_preds)
```

```
multistep_preds = d2l.zeros(data.T)
multistep_preds = multistep_preds.at[:].set(data.x)
for i in range(data.num_train + data.tau, data.T):
    pred = model.apply({'params': trainer.state.params},
                       d2l.reshape(multistep_preds[i-data.tau : i], (1, -1)))
    multistep_preds = multistep_preds.at[i].set(pred.item())
```

```
multistep_preds = tf.Variable(d2l.zeros(data.T))
multistep_preds[:].assign(data.x)
for i in range(data.num_train + data.tau, data.T):
    multistep_preds[i].assign(d2l.reshape(model(
        d2l.reshape(multistep_preds[i-data.tau : i], (1, -1))), ()))
```

```
d2l.plot([data.time[data.tau:], data.time[data.num_train+data.tau:]],
         [onestep_preds, multistep_preds[data.num_train+data.tau:]], 'time',
         'x', legend=['1-step preds', 'multistep preds'], figsize=(6, 3))
```



残念ながら、この場合は見事に失敗する。予測は数ステップ後にはかなり急速に一定値へと収束してしまう。なぜ、さらに先を予測するときにアルゴリズムの性能はこれほど悪化したのだろうか。結局のところ、誤差が蓄積するからである。ステップ 1 の後にある誤差 $\epsilon_1 = \bar{\epsilon}$ があるとしよう。すると、ステップ 2 の 入力は ϵ_1 によって摂動されるので、ある定数 c に対して $\epsilon_2 = \bar{\epsilon} + c\epsilon_1$ 程度の誤差を被る。以下同様である。予測は真の観測値から急速に乖離しうるのである。あなたもこのよくある現象に見覚えがあるかもしれない。たとえば、24 時間先までの天気予報はかなり正確な傾向があるが、それを超えると精度は急速に低下する。この章および以降では、これを改善する方法について議論する。

$k = 1, 4, 16, 64$ に対してシーケンス全体で予測を計算し、k ステップ先予測の難しさを詳しく見てみよう。

```
def k_step_pred(k):
    features = []
    for i in range(data.tau):
        features.append(data.x[i : i+data.T-data.tau-k+1])
    # The (i+tau)-th element stores the (i+1)-step-ahead predictions
    for i in range(k):
        preds = model(d2l.stack(features[i : i+data.tau], 1))
        features.append(d2l.reshape(preds, -1))
    return features[data.tau:]
```

```
def k_step_pred(k):
    features = []
    for i in range(data.tau):
        features.append(data.x[i : i+data.T-data.tau-k+1])
    # The (i+tau)-th element stores the (i+1)-step-ahead predictions
    for i in range(k):
        preds = model(d2l.stack(features[i : i+data.tau], 1))
        features.append(d2l.reshape(preds, -1))
    return features[data.tau:]
```

```
def k_step_pred(k):
    features = []
    for i in range(data.tau):
        features.append(data.x[i : i+data.T-data.tau-k+1])
    # The (i+tau)-th element stores the (i+1)-step-ahead predictions
    for i in range(k):
        preds = model.apply({'params': trainer.state.params},
                            d2l.stack(features[i : i+data.tau], 1))
        features.append(d2l.reshape(preds, -1))
    return features[data.tau:]
```

```
def k_step_pred(k):
    features = []
    for i in range(data.tau):
        features.append(data.x[i : i+data.T-data.tau-k+1])
    # The (i+tau)-th element stores the (i+1)-step-ahead predictions
```

(continues on next page)

(continued from previous page)

```
    for i in range(k):
        preds = model(d2l.stack(features[i : i+data.tau], 1))
        features.append(d2l.reshape(preds, -1))
    return features[data.tau:]
```

```
steps = (1, 4, 16, 64)
preds = k_step_pred(steps[-1])
d2l.plot(data.time[data.tau+steps[-1]-1:],
         [d2l.numpy(preds[k-1]) for k in steps], 'time', 'x',
         legend=[f'{k}-step preds' for k in steps], figsize=(6, 3))
```

これは、さらに先を予測しようとすると予測品質がどのように変化するかを明確に示している。4 ステップ先予測はまだ良さそうに見えるが、それを超えるとほとんど役に立たない。

12.1.5 要約

補間と外挿の難しさにはかなりの差がある。したがって、シーケンスがある場合は、訓練時にデータの時間順序を必ず尊重し、つまり未来のデータで訓練してはならない。この種のデータでは、シーケンスモデルの推定には特殊な統計的手法が必要である。よく使われる選択肢は、自己回帰モデルと潜在変数自己回帰モデルである。因果モデル（たとえば時間が前に進む場合）では、前向きの方向を推定するほうが逆方向より通常ずっと容易である。時刻 t までの観測シーケンスに対して、時刻 $t+k$ における予測出力を k ステップ先予測と呼ぶ。k を増やして時間的にさらに先を予測するほど、誤差は蓄積し、予測品質はしばしば劇的に低下する。

12.1.6 演習

1. この節の実験のモデルを改善せよ。
    1. 過去 4 観測より多くを取り入れるだろうか。実際にはどれくらい必要だろうか。
    2. ノイズがなければ、過去の観測は何個必要だろうか。ヒント：sin と cos は微分方程式として書ける。
    3. 総特徴量数を一定に保ちながら、より古い観測を取り入れられるだろうか。それで精度は向上するだろうか。なぜか。
    4. ニューラルネットワークのアーキテクチャを変えて性能を評価せよ。新しいモデルはより多くのエポックで訓練してもよい。何が観察されるだろうか。
2. 投資家が買うべき有望な証券を見つけたいと考えている。どれがうまくいきそうかを判断するために、過去のリターンを見る。何がこの戦略でうまくいかなくなる可能性があるだろうか。

3. 因果性はテキストにも当てはまるだろうか。どの程度までだろうか。
4. データのダイナミクスを捉えるために、潜在自己回帰モデルが必要になる例を挙げよ。

12.2 生テキストを系列データに変換する

この本全体を通して、私たちはしばしば、単語、文字、または単語片の系列として表現されたテキストデータを扱う。まずは、生テキストを適切な形式の系列へ変換するための基本的な道具が必要である。典型的な前処理パイプラインは次の手順を実行する。

1. テキストを文字列としてメモリに読み込む。
2. 文字列をトークン（例：単語や文字）に分割する。
3. 各語彙要素を数値インデックスに対応付けるための語彙辞書を構築する。
4. テキストを数値インデックスの系列に変換する。

```
%load_ext d2lbook.tab
tab.interact_select(['mxnet', 'pytorch', 'tensorflow', 'jax'])
```

```
import collections
import re
from d2l import torch as d2l
import torch
import random
```

```
import collections
import re
from d2l import mxnet as d2l
from mxnet import np, npx
import random
npx.set_np()
```

```
import collections
from d2l import jax as d2l
import jax
from jax import numpy as jnp
import random
import re
```

```
import collections
import re
```

(continues on next page)

(continued from previous page)

```
from d2l import tensorflow as d2l
import tensorflow as tf
import random
```

12.2.1 データセットの読み込み

78 

ここでは、H. G. Wells の『タイムマシン』[78] を扱う。この本は 3 万語強から成る。実際のアプリケーションでは通常、はるかに大規模なデータセットを扱うが、前処理パイプラインを示すにはこれで十分である。以下の _download メソッドは生テキストを文字列として読み込む。

```
class TimeMachine(d2l.DataModule): #@save
    """The Time Machine dataset."""
    def _download(self):
        fname = d2l.download(d2l.DATA_URL + 'timemachine.txt', self.root,
                             '090b5e7e70c295757f55df93cb0a180b9691891a')
        with open(fname) as f:
            return f.read()

data = TimeMachine()
raw_text = data._download()
raw_text[:60]
```

```
Downloading ../data/timemachine.txt from http://d2l-data.s3-accelerate.
↪amazonaws.com/timemachine.txt...
```

```
'The Time Machine, by H. G. Wells [1898]\n\n\nThe Time Tra'
```

簡単のため、生テキストの前処理では句読点と大文字小文字を無視する。

```
@d2l.add_to_class(TimeMachine)  #@save
def _preprocess(self, text):
    return re.sub('[^A-Za-z]+', ' ', text).lower()

text = data._preprocess(raw_text)
text[:60]
```

```
'the time machine by h g wells i the time traveller for so it'
```

12.2.2 トークン化

トークンとは、テキストの原子的（不可分な）単位である。各タイムステップは 1 つのトークンに対応するが、何を正確にトークンとみなすかは設計上の選択である。たとえば、文“Baby needs a new pair of shoes”を 7 語の系列として表現することもできる。この場合、すべての単語の集合は大きな語彙（通常は数万語から数十万語）を構成する。あるいは、同じ文を 30 文字からなる、より長い系列として表現することもでき、その場合ははるかに小さな語彙を使う（ASCII

文字は 256 種類しかない）。以下では、前処理済みテキストを文字の系列にトークン化する。

```
@d2l.add_to_class(TimeMachine)  #@save
def _tokenize(self, text):
    return list(text)

tokens = data._tokenize(text)
','.join(tokens[:30])
```

```
't,h,e, ,t,i,m,e, ,m,a,c,h,i,n,e, ,b,y, ,h, ,g, ,w,e,l,l,s, '
```

12.2.3 語彙

これらのトークンはまだ文字列である。しかし、モデルへの入力は最終的には数値入力でなければならない。次に、語彙を構築するためのクラスを導入する。すなわち、各異なるトークン値を一意のインデックスに対応付けるオブジェクトである。まず、訓練 コーパスに含まれる一意なトークンの集合を求める。次に、各一意トークンに数値インデックスを割り当てる。まれな語彙要素は、便宜上しばしば除外される。訓練時またはテスト時に、以前に見たことがないトークンや語彙から除外されたトークンに遭遇した場合は、特別な“<unk>”トークンで表し、これが 未知の値であることを示す。

```
class Vocab:  #@save
    """Vocabulary for text."""
    def __init__(self, tokens=[], min_freq=0, reserved_tokens=[]):
        # Flatten a 2D list if needed
        if tokens and isinstance(tokens[0], list):
            tokens = [token for line in tokens for token in line]
        # Count token frequencies
        counter = collections.Counter(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                  reverse=True)
        # The list of unique tokens
        self.idx_to_token = list(sorted(set(['<unk>'] + reserved_tokens + [
            token for token, freq in self.token_freqs if freq >= min_freq])))
        self.token_to_idx = {token: idx
                             for idx, token in enumerate(self.idx_to_token)}

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
        if hasattr(indices, '__len__') and len(indices) > 1:
            return [self.idx_to_token[int(index)] for index in indices]
        return self.idx_to_token[indices]
```

(continues on next page)

(continued from previous page)

```
    @property
    def unk(self):  # Index for the unknown token
        return self.token_to_idx['<unk>']
```

ここで、データセットのための語彙を構築し、文字列の系列を数値インデックスのリストに変換する。情報は失われておらず、データセットを元の（文字列）表現に簡単に戻せることに注意せよ。

```
vocab = Vocab(tokens)
indices = vocab[tokens[:10]]
print('indices:', indices)
print('words:', vocab.to_tokens(indices))
```

```
indices: [21, 9, 6, 0, 21, 10, 14, 6, 0, 14]
words: ['t', 'h', 'e', ' ', 't', 'i', 'm', 'e', ' ', 'm']
```

12.2.4 まとめて扱う

上のクラスとメソッドを用いて、以下の `TimeMachine` クラスの `build` メソッドにすべてをまとめる。このメソッドは、トークンインデックスのリストである `corpus` と、『タイムマシン』コーパスの語彙である `vocab` を返す。ここで行った変更は次のとおりである。(i) 後の節での学習を簡単にするため、テキストを単語ではなく文字にトークン化する。(ii) `corpus` はトークンリストのリストではなく単一のリストである。これは、『タイムマシン』データセットの各テキスト行が必ずしも文や段落ではないためである。

```
@d2l.add_to_class(TimeMachine)  #@save
def build(self, raw_text, vocab=None):
    tokens = self._tokenize(self._preprocess(raw_text))
    if vocab is None: vocab = Vocab(tokens)
    corpus = [vocab[token] for token in tokens]
    return corpus, vocab

corpus, vocab = data.build(raw_text)
len(corpus), len(vocab)
```

```
(173428, 28)
```

12.2.5 探索的な言語統計

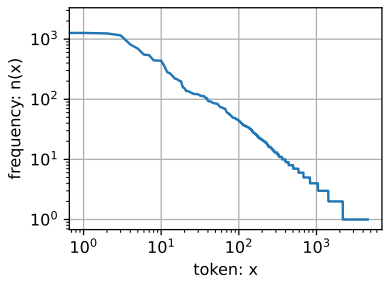
実際のコーパスと、単語に対して定義した `Vocab` クラスを用いると、コーパス内での単語の使われ方に関する基本統計を調べられる。以下では、『タイムマシン』で使われている単語から語彙を構築し、最も頻出する 10 語を表示する。

```
words = text.split()
vocab = Vocab(words)
vocab.token_freqs[:10]
```

```
[('the', 2261),
 ('i', 1267),
 ('and', 1245),
 ('of', 1155),
 ('a', 816),
 ('to', 695),
 ('was', 552),
 ('in', 541),
 ('that', 443),
 ('my', 440)]
```

最も頻出する 10 語は、あまり説明的ではないことに注意せよ。任意の本を選んでも、非常によく似たリストが得られるのではないかと想像できるかもしれない。“the”や“a”のような冠詞、“i”や“my”のような代名詞、“of”、“to”、“in”のような前置詞は、一般的な統語的役割を果たすため頻繁に現れる。このように、一般的だが特に説明的ではない単語はしばしばストップワードと呼ばれ、いわゆる bag-of-words 表現に基づく従来世代のテキスト分類器では、たいてい除外されていた。しかし、それらは意味を持っており、現代の RNN や Transformer ベースのニューラルモデルを扱う際に必ずしも除外する必要はない。さらにリストの下の方を見ると、単語頻度が急速に減衰していることがわかる。10^{th} に頻出する単語は、最頻出単語の 1/5 未満しか現れない。単語頻度は、順位が下がるにつれてべき乗則分布（具体的には Zipf 分布）に従う傾向がある。よりよく理解するために、単語頻度の図を描く。

```
freqs = [freq for token, freq in vocab.token_freqs]
d2l.plot(freqs, xlabel='token: x', ylabel='frequency: n(x)',
         xscale='log', yscale='log')
```



最初の数語を例外として扱えば、残りのすべての単語は、対数–対数プロット上でおおむね直線に従う。この現象は *Zipf* の法則によって捉えられる。これは、i^{th} に頻出する単語の頻度 n_i が次を満たすことを述べている。

$$n_i \propto \frac{1}{i^\alpha}, \tag{12.2.1}$$

これは次と同値である。

$$\log n_i = -\alpha \log i + c, \tag{12.2.2}$$

ここで α は分布を特徴づける指数であり、c は定数である。単語を出現回数の統計でモデル化しようとするなら、ここで立ち止まって考えるべきだろう。結局のところ、頻度の低い単語としても知られるテール部分の頻度を大きく過大評価してしまうからである。では、2 語連続の組（bigram）、3 語連続の組（trigram）など、他の単語の組み合わせはどうだろうか。さらに先まで見てみよう。bigram の頻度が単一語（unigram）の頻度と同じように振る舞うかを確認する。

```
bigram_tokens = ['--'.join(pair) for pair in zip(words[:-1], words[1:])]
bigram_vocab = Vocab(bigram_tokens)
bigram_vocab.token_freqs[:10]
```

```
[('of--the', 309),
 ('in--the', 169),
 ('i--had', 130),
 ('i--was', 112),
 ('and--the', 109),
 ('the--time', 102),
 ('it--was', 99),
 ('to--the', 85),
 ('as--i', 78),
 ('of--a', 73)]
```

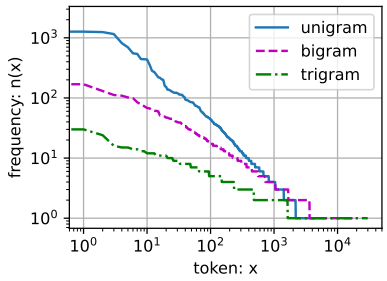
ここで注目すべき点が 1 つある。最頻出の 10 個の語の組のうち 9 個はストップワード同士から成り、実際の本に関係するものは“the time”だけである。さらに、trigram の頻度も同じように振る舞うかを見てみよう。

```
trigram_tokens = ['--'.join(triple) for triple in zip(
    words[:-2], words[1:-1], words[2:])]
trigram_vocab = Vocab(trigram_tokens)
trigram_vocab.token_freqs[:10]
```

```
[('the--time--traveller', 59),
 ('the--time--machine', 30),
 ('the--medical--man', 24),
 ('it--seemed--to', 16),
 ('it--was--a', 15),
 ('here--and--there', 15),
 ('seemed--to--me', 14),
 ('i--did--not', 14),
 ('i--saw--the', 13),
 ('i--began--to', 13)]
```

では、これら 3 つのモデル、すなわち unigram、bigram、trigram におけるトークン頻度を可視化してみよう。

```
bigram_freqs = [freq for token, freq in bigram_vocab.token_freqs]
trigram_freqs = [freq for token, freq in trigram_vocab.token_freqs]
d2l.plot([freqs, bigram_freqs, trigram_freqs], xlabel='token: x',
         ylabel='frequency: n(x)', xscale='log', yscale='log',
         legend=['unigram', 'bigram', 'trigram'])
```



この図は非常に興味深いものである。第一に、unigram の単語だけでなく、単語列も Zipf の法則に従っているように見えるが、系列長に応じて (12.2.1) の指数 α は小さくなる。第二に、異なる n-gram の数はそれほど多くない。これは、言語にはかなり多くの構造があることを示唆している。第三に、多くの n-gram は非常にまれにしか現れない。このことは、ある種の手法が言語モデリングに適さないことを意味し、深層学習モデルの利用を動機づける。これについては次節で議論する。

12.2.6 まとめ

テキストは、深層学習で遭遇する最も一般的な系列データ形式の 1 つである。トークンとして何を採用するかの一般的な選択肢は、文字、単語、単語片である。テキストを前処理するには、通常、(i) テキストをトークンに分割し、(ii) トークン文字列を数値インデックスに写像する語彙を構築し、(iii) モデルが扱えるようにテキストデータをトークンインデックスに変換する。実際には、単語頻度は Zipf の法則に従う傾向がある。これは個々の単語（unigram）だけでなく、n-gram に対しても成り立つ。

12.2.7 演習

1. この節の実験では、テキストを単語にトークン化し、`Vocab` インスタンスの `min_freq` 引数の値を変えてみよ。`min_freq` の変化が、結果として得られる語彙サイズにどのように影響するかを定性的に述べよ。
2. このコーパスにおける unigram、bigram、trigram の Zipf 分布の指数を推定せよ。
3. 他のデータソースを見つけよ（標準的な機械学習データセットをダウンロードする、別のパブリックドメインの本を選ぶ、Web サイトをスクレイピングする、など）。それぞれについて、単語レベルと文字レベルの両方でデータをトークン化せよ。`min_freq` の同じ値に対して、『タイムマシン』コーパス

と語彙サイズを比較するとどうなるか。これらのコーパスに対する unigram と bigram の分布に対応する Zipf 分布の指数を推定せよ。『タイムマシン』コーパスで観測した値と比べてどうか。

12.3 言語モデル

Section 12.2 では、テキスト系列をトークンへ写像する方法を見た。ここでこれらのトークンは、単語や文字のような離散的な観測の系列としてみなせる。長さ T のテキスト系列のトークンを順に $x_1, x_2, \ldots, x_T$ とする。言語モデルの目標は、系列全体の同時確率を推定することである。

$$P(x_1, x_2, \ldots, x_T), \tag{12.3.1}$$

ここで Section 12.1 の統計的手法を適用できる。

言語モデルは非常に有用である。たとえば、理想的な言語モデルは、1 トークンずつ $x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1)$ とサンプリングするだけで、自ら自然なテキストを生成できるはずである。タイプライターを打つサルとはまったく異なり、このようなモデルから出てくるテキストはすべて自然言語、たとえば英語として通用するだろう。さらに、過去の対話断片を条件にするだけで、意味のある対話を生成するのにも十分である。明らかに、私たちはそのようなシステムの設計からはまだほど遠い。なぜなら、それには単に文法的にもっともらしい内容を生成するだけでなく、テキストを 理解する必要があるからである。

それでも、言語モデルは制約された形であっても大いに役立つ。たとえば、「to recognize speech」と「to wreck a nice beach」は非常によく似た響きを持つ。これは音声認識において曖昧性を生みうるが、言語モデルを用いれば、後者の変換を突飛なものとして棄却することで容易に解決できる。同様に、文書要約アルゴリズムでは、「dog bites man」が「man bites dog」よりはるかに頻出であることや、「I want to eat grandma」がかなり不穏な表現である一方、「I want to eat, grandma」ははるかに無害であることを知っていると有益である。

```
%load_ext d2lbook.tab
tab.interact_select(['mxnet', 'pytorch', 'tensorflow', 'jax'])
```

```
from d2l import torch as d2l
import torch
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()
```

```
from d2l import jax as d2l
from jax import numpy as jnp
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

12.3.1 言語モデルの学習

当然の疑問は、文書、あるいはトークン系列そのものをどのようにモデル化すべきかである。テキストデータを単語レベルでトークン化するとしよう。まずは基本的な確率の法則を適用してみる。

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}). \tag{12.3.2}$$

たとえば、4 語からなるテキスト系列の確率は次のように表される。

$$\begin{aligned} &P(\text{deep}, \text{learning}, \text{is}, \text{fun}) \\ =&P(\text{deep})P(\text{learning} \mid \text{deep})P(\text{is} \mid \text{deep}, \text{learning})P(\text{fun} \mid \text{deep}, \text{learning}, \text{is}). \end{aligned} \tag{12.3.3}$$

マルコフモデルと n-gram

Section 12.1 における系列モデルの分析の中で、言語モデリングにマルコフモデルを適用してみよう。系列上の分布が 1 次のマルコフ性 $P(x_{t+1} \mid x_t, \ldots, x_1) = P(x_{t+1} \mid x_t)$ を満たすとする。高次の場合は、より長い依存関係に対応する。これにより、系列をモデル化するために適用できるいくつかの近似が得られる。

$$\begin{aligned} P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2)P(x_3)P(x_4), \\ P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_2)P(x_4 \mid x_3), \\ P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_1, x_2)P(x_4 \mid x_2, x_3). \end{aligned} \tag{12.3.4}$$

1 変数、2 変数、3 変数を含む確率式は、それぞれ通常 *unigram*、*bigram*、*trigram* モデルと呼ばれる。言語モデルを計算するには、単語の確率と、直前のいくつかの単語が与えられたときの単語の条件付き確率を計算する必要がある。注意すべきなのは、このような確率が言語モデルのパラメータであるということである。

単語頻度

ここでは、学習データセットが、Wikipedia の全項目や Project Gutenberg[79] 、およびウェブ上に投稿されたすべてのテキストのような、大規模なテキストコーパスであると仮定する。単語の確率は、学習データセットにおける対象単語の相対頻度から計算できる。たとえば、推定値 $\hat{P}(\text{deep})$ は、「deep」で始まる任意の文の確率として計算できる。やや精度の低い方法としては、「deep」という単語の出現回数をすべて数え、それをコーパス中の総単語数で割る方法がある。

79 

これはかなりうまく機能し、とくに頻出語では有効である。さらに進めると、次を推定できる。

$$\hat{P}(\text{learning} \mid \text{deep}) = \frac{n(\text{deep, learning})}{n(\text{deep})}, \tag{12.3.5}$$

ここで $n(x)$ と $n(x, x')$ は、それぞれ単独の単語と連続する単語対の出現回数である。残念ながら、単語対の確率を推定するのはやや難しい。というのも、「deep learning」の出現頻度はずっと低いからである。とくに、珍しい単語の組み合わせでは、十分な出現回数を見つけて正確な推定を得るのが難しい場合がある。Section 12.2.5 の実証結果が示唆するように、3 語以上の組み合わせでは状況はさらに悪化する。データセット中で見かけないであろう、もっともらしい 3 語の組み合わせは数多く存在する。こうした単語の組み合わせに非ゼロのカウントを割り当てる解決策を用意しない限り、言語モデルでそれらを使うことはできない。データセットが小さい場合や単語が非常に稀な場合には、そのうち 1 つも見つからないことさえある。

ラプラス平滑化

一般的な戦略の 1 つは、何らかの形の ラプラス平滑化を行うことである。解決策は、すべてのカウントに小さな定数を加えることである。学習集合中の単語総数を n、異なる単語数を m とする。この解決策は単独語に対して有効であり、たとえば次のように表せる。

$$\begin{aligned}
\hat{P}(x) &= \frac{n(x) + \epsilon_1/m}{n + \epsilon_1}, \\
\hat{P}(x' \mid x) &= \frac{n(x, x') + \epsilon_2 \hat{P}(x')}{n(x) + \epsilon_2}, \\
\hat{P}(x'' \mid x, x') &= \frac{n(x, x', x'') + \epsilon_3 \hat{P}(x'')}{n(x, x') + \epsilon_3}.
\end{aligned} \tag{12.3.6}$$

ここで $\epsilon_1, \epsilon_2,$ および ϵ_3 はハイパーパラメータである。ϵ_1 を例に取ると、$\epsilon_1 = 0$ のとき平滑化は行われない。ϵ_1 が正の無限大に近づくと、$\hat{P}(x)$ は一様確率 $1/m$ に近づく。上記は、他の手法が実現できること (Wood *et al.*, 2011) のかなり原始的な変種である。

残念ながら、このようなモデルは次の理由からすぐに扱いにくくなる。第一に、Section 12.2.5 で述べたように、多くの n-gram は非常に稀にしか現れず、ラプラス平滑化は言語モデリングにはあまり適していない。第二に、すべてのカウントを保存する必要がある。第三に、これは単語の意味を完全に無視している。たとえば、「cat」と「feline」は関連する文脈で現れるべきである。このようなモデルを追加の文脈に合わせて調整するのはかなり難しいが、深層学習ベースの言語モデルはこれを取り込むのに適している。最後に、長い単語系列はほぼ確実に新規のものになるため、過去に見た単語系列の頻度を単に数えるだけのモデルはそこでうまく機能しない。したがって、本章の残りでは、言語モデリングにニューラルネットワークを用いることに焦点を当てる。

12.3.2 困惑度

次に、言語モデルの品質をどのように測るかを議論し、その後の節でモデルを評価するために用いる。1 つの方法は、テキストがどれだけ意外かを調べることである。良い言語モデルは、次に来るトークンを高い精度で予測できる。異なる言語モデルが提案する「It is raining」の続きとして、次の例を考えよう。

1. “It is raining outside”

2. “It is raining banana tree”

3. “It is raining piouw;kcj pwepoiut”

品質の観点では、例 1 が明らかに最良である。単語は意味をなし、論理的にも整合している。意味的にどの単語が続くかを完全に正確に反映しているわけではないかもしれないが（「in San Francisco」や「in winter」でもまったく妥当な続きである）、モデルはどの種類の単語が続くかを捉えることができている。例 2 は、無意味な続きが生成されるため、かなり悪い。それでも少なくとも、モデルは単語の綴り方と、単語間のある程度の相関を学習している。最後に、例 3 は、データにうまく適合していない、訓練不十分なモデルを示している。

系列の尤度を計算することでモデルの品質を測ることもできる。残念ながら、これは理解しにくく比較もしづらい数値である。そもそも、短い系列のほうが長い系列よりはるかに起こりやすいので、トルストイの大作 *War and Peace* でモデルを評価すると、たとえばサン＝テグジュペリの中編 *The Little Prince* で評価した場合より、尤度は必然的にずっと小さくなる。欠けているのは平均に相当するものである。

ここで情報理論が役立つ。ソフトマックス回帰を導入したときに、エントロピー、驚き、クロスエントロピーを定義した（Section 7.1.3）。テキストを圧縮したいなら、現在のトークン集合が与えられたときに次のトークンを予測することを考えればよい。より良い言語モデルほど、次のトークンをより正確に予測できる。したがって、系列を圧縮するのに必要なビット数をより少なくできる。そこで、系列の n 個のトークン全体で平均したクロスエントロピー損失として測定できる。

$$\frac{1}{n}\sum_{t=1}^{n} -\log P(x_t \mid x_{t-1}, \ldots, x_1), \tag{12.3.7}$$

ここで P は言語モデルによって与えられ、x_t は時刻 t に系列から観測された実際のトークンである。これにより、異なる長さの文書に対する性能を比較可能にできる。歴史的な理由から、自然言語処理の研究者は *perplexity* と呼ばれる量を用いることを好む。要するに、これは (12.3.7) の指数である。

$$\exp\left(-\frac{1}{n}\sum_{t=1}^{n} \log P(x_t \mid x_{t-1}, \ldots, x_1)\right). \tag{12.3.8}$$

困惑度は、次にどのトークンを選ぶかを決めるときに実際に取りうる選択肢の

数の幾何平均の逆数として理解するのが最もよい。いくつかの場合を見てみよう。

- 最良の場合、モデルは常に目標トークンの確率を 1 と完全に推定する。このときモデルの困惑度は 1 である。
- 最悪の場合、モデルは常に目標トークンの確率を 0 と予測する。このとき困惑度は正の無限大である。
- 基準として、モデルは語彙中の利用可能なすべてのトークンに対して一様分布を予測する。この場合、困惑度は語彙の異なるトークン数に等しい。実際、系列を何の圧縮もせずに保存するなら、これが符号化のためにできる最善である。したがって、これは有用なモデルが必ず上回らなければならない、非自明な上界を与える。

12.3.3 系列の分割

ここでは、ニューラルネットワークを用いて言語モデルを設計し、困惑度を使って、テキスト系列において現在のトークン集合が与えられたときに次のトークンを予測する性能がどれほど良いかを評価する。モデルを導入する前に、それがあらかじめ定められた長さの系列のミニバッチを一度に処理すると仮定しよう。ここでの問題は、入力系列と目標系列のミニバッチをランダムに読み込むにはどうすればよいか、である。

データセットが corpus にある T 個のトークンインデックスからなる系列の形をとるとしよう。これを各部分系列が n 個のトークン（時刻）を持つような部分系列に分割する。各エポックでデータセット全体の（ほぼ）すべてのトークンを走査し、長さ n のすべての可能な部分系列を得るために、ランダム性を導入できる。より具体的には、各エポックの最初に、最初の d 個のトークンを捨てる。ここで $d \in [0, n)$ は一様にランダムサンプリングされる。その後、残りの系列を $m = \lfloor (T - d)/n \rfloor$ 個の部分系列に分割する。時刻 t でトークン x_t から始まる長さ n の部分系列を $\mathbf{x}_t = [x_t, \ldots, x_{t+n-1}]$ と表す。すると、分割された m 個の部分系列は $\mathbf{x}_d, \mathbf{x}_{d+n}, \ldots, \mathbf{x}_{d+n(m-1)}$. 各部分系列は、言語モデルへの入力系列として用いられる。

言語モデリングでは、これまで見てきたトークンに基づいて次のトークンを予測することが目標である。したがって、目標（ラベル）は元の系列を 1 トークンずらしたものである。任意の入力系列 $\mathbf{x}_t$ に対する目標系列は、長さ n の $\mathbf{x}_{t+1}$ である。

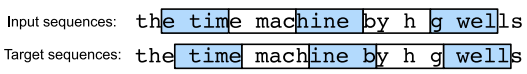


Fig. 12.3.1 分割された長さ 5 の部分系列から 5 組の入力系列と目標系列を得る。

Fig. 12.3.1 は、$n = 5$ と $d = 2$ のときに 5 組の入力系列と目標系列を得る例を示している。

```
@d2l.add_to_class(d2l.TimeMachine)  #@save
def __init__(self, batch_size, num_steps, num_train=10000, num_val=5000):
    super(d2l.TimeMachine, self).__init__()
    self.save_hyperparameters()
    corpus, self.vocab = self.build(self._download())
    array = d2l.tensor([corpus[i:i+num_steps+1]
                        for i in range(len(corpus)-num_steps)])
    self.X, self.Y = array[:,:-1], array[:,1:]
```

言語モデルを訓練するために、入力系列と目標系列の組をミニバッチでランダムにサンプリングする。以下のデータローダーは、データセットから毎回ランダムにミニバッチを生成する。引数 batch_size は各ミニバッチ中の部分系列例の数を指定し、num_steps はトークン単位の部分系列長である。

```
@d2l.add_to_class(d2l.TimeMachine)  #@save
def get_dataloader(self, train):
    idx = slice(0, self.num_train) if train else slice(
        self.num_train, self.num_train + self.num_val)
    return self.get_tensorloader([self.X, self.Y], train, idx)
```

以下でわかるように、目標系列のミニバッチは入力系列を 1 トークンずらすことで得られる。

```
data = d2l.TimeMachine(batch_size=2, num_steps=10)
for X, Y in data.train_dataloader():
    print('X:', X, '\nY:', Y)
    break
```

```
X: tensor([[16,  4, 21,  2,  8, 16, 15,  2, 13,  0],
        [ 0,  8, 19,  2, 23, 10, 21,  2, 21, 10]])
Y: tensor([[ 4, 21,  2,  8, 16, 15,  2, 13,  0, 21],
        [ 8, 19,  2, 23, 10, 21,  2, 21, 10, 16]])
```

12.3.4 要約と考察

言語モデルはテキスト系列の同時確率を推定する。長い系列に対しては、n-gram は依存関係を切り詰めることで便利なモデルを提供する。しかし、構造は多いのに頻度が十分でないため、ラプラス平滑化で稀な単語の組み合わせを効率よく扱うのは難しい。したがって、以降の節ではニューラル言語モデリングに焦点を当てる。言語モデルを訓練するには、入力系列と目標系列の組をミニバッチでランダムにサンプリングできる。訓練後は、困惑度を用いて言語モデルの品質を測定する。

言語モデルは、データ量、モデルサイズ、訓練計算量を増やすことでスケールアップできる。大規模言語モデルは、入力テキストの指示に基づいて出力テキストを予測することで、望ましいタスクを実行できる。後で議論するように（たとえば Section 14.9）、現時点では大規模言語モデルが、多様なタスクにおける最先端システムの基盤を成している。

12.3.5 演習

1. 学習データセットに 10 万語があると仮定する。4-gram はどれだけの単語頻度と隣接する複数語の頻度を保存する必要があるか。
2. 対話をどのようにモデル化すべきか。
3. 長い系列データを読み込むための他の方法として、どのようなものが考えられるか。
4. 各エポックの最初に、最初の数個のトークンを一様ランダムな数だけ捨てる方法を考えよ。
   1. これは文書上の系列に対して、本当に完全に一様な分布をもたらすだろうか。
   2. さらに一様にするには何をすべきだろうか。
5. ある系列例を完全な文にしたい場合、ミニバッチサンプリングではどのような問題が生じるだろうか。どうすれば修正できるだろうか。

12.4 再帰型ニューラルネットワーク

Section 12.3 では、言語モデリングのためのマルコフモデルと n-gram を説明した。そこでは、時刻 t におけるトークン x_t の条件付き確率は、直前の $n-1$ 個のトークンのみに依存する。時刻 $t-(n-1)$ より前のトークンが x_t に及ぼしうる影響を取り込みたい場合は、n を大きくする必要がある。しかし、その場合、モデルパラメータの数もそれに伴って指数的に増加する。というのも、語彙集合 $\mathcal{V}$ に対して $|\mathcal{V}|^n$ 個の数を保持しなければならないからである。したがって、$P(x_t \mid x_{t-1}, \ldots, x_1)$ を直接モデル化するよりも、潜在変数モデルを用いる方が望ましい。

$$P(x_t \mid x_{t-1}, \ldots, x_1) \approx P(x_t \mid h_{t-1}), \tag{12.4.1}$$

ここで h_{t-1} は、時刻 $t-1$ までの系列情報を保持する 隠れ状態である。一般に、任意の時刻 t における隠れ状態は、現在の入力 x_t と前の隠れ状態 h_{t-1} の両方に基づいて計算できる。

$$h_t = f(x_t, h_{t-1}). \tag{12.4.2}$$

(12.4.2) において十分に強力な関数 f を用いれば、この潜在変数モデルは近似ではない。結局のところ、h_t はこれまでに観測したすべてのデータを単に保持していてもよいからである。しかし、その場合、計算と記憶の両方が高コストになる可能性がある。

Section 8 で、隠れユニットを持つ隠れ層について説明したことを思い出してほしい。ここで重要なのは、隠れ層と隠れ状態は、まったく異なる概念だというこ

とである。隠れ層は、説明したように、入力から出力へ至る経路の途中で外から見えない層である。一方、隠れ状態は技術的には、ある時点で行う処理に対する 入力であり、それは過去の時刻のデータを見て初めて計算できる。

再帰型ニューラルネットワーク（RNN）は、隠れ状態を持つニューラルネットワークである。RNN モデルを導入する前に、まず Section 8.1 で導入した MLP モデルを振り返ろう。

```
%load_ext d2lbook.tab
tab.interact_select('mxnet', 'pytorch', 'tensorflow', 'jax')
```

```
from d2l import torch as d2l
import torch
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()
```

```
from d2l import jax as d2l
import jax
from jax import numpy as jnp
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

12.4.1 隠れ状態を持たないニューラルネットワーク

単一の隠れ層を持つ MLP を見てみよう。隠れ層の活性化関数を ϕ とする。バッチサイズが n、入力次元が d のミニバッチの例 $\mathbf{X} \in \mathbb{R}^{n \times d}$ が与えられたとき、隠れ層の出力 $\mathbf{H} \in \mathbb{R}^{n \times h}$ は次のように計算される。

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{\text{xh}} + \mathbf{b}_{\text{h}}). \tag{12.4.3}$$

(12.4.3) では、隠れ層に対して重みパラメータ $\mathbf{W}_{\text{xh}} \in \mathbb{R}^{d \times h}$、バイアスパラメータ $\mathbf{b}_{\text{h}} \in \mathbb{R}^{1 \times h}$、および隠れユニット数 h を用いている。このため、加算の際にはブロードキャスト（Section 5.1.4 を参照）を適用する。次に、隠れ層の出力 $\mathbf{H}$ を出力層の入力として用いる。出力層は次式で与えられる。

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}, \tag{12.4.4}$$

ここで $\mathbf{O} \in \mathbb{R}^{n \times q}$ は出力変数、$\mathbf{W}_{\text{hq}} \in \mathbb{R}^{h \times q}$ は重みパラメータ、$\mathbf{b}_{\text{q}} \in \mathbb{R}^{1 \times q}$ は出力層のバイアスパラメータである。分類問題であれば、$\text{softmax}(\mathbf{O})$ を用いて出力カテゴリの確率分布を計算できる。

これは Section 12.1 で以前に解いた回帰問題と完全に同様なので、詳細は省略する。要するに、特徴とラベルのペアをランダムに取り出し、自動微分と確率

的勾配降下法によってネットワークのパラメータを学習できるということである。

12.4.2 隠れ状態を持つ再帰型ニューラルネットワーク

隠れ状態がある場合は、事情がまったく異なる。構造をもう少し詳しく見てみよう。

時刻 t において、ミニバッチの入力 $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ があると仮定する。言い換えると、n 個の系列例からなるミニバッチについて、$\mathbf{X}_t$ の各行は系列中の時刻 t における 1 つの例に対応する。次に、時刻 t の隠れ層出力を $\mathbf{H}_t \in \mathbb{R}^{n\times h}$ と表す。MLP とは異なり、ここでは前の時刻の隠れ層出力 $\mathbf{H}_{t-1}$ を保持し、前の時刻の隠れ層出力を現在の時刻でどのように使うかを表す新しい重みパラメータ $\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h\times h}$ を導入する。具体的には、現在の時刻の隠れ層出力の計算は、現在の時刻の入力と前の時刻の隠れ層出力の両方によって決まる。

$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}). \tag{12.4.5}$$

(12.4.3) と比べると、(12.4.5) では項 $\mathbf{H}_{t-1}\mathbf{W}_{\text{hh}}$ が 1 つ追加されており、そのため (12.4.2) を具体化している。隣接する時刻の隠れ層出力 $\mathbf{H}_t$ と $\mathbf{H}_{t-1}$ の関係から、これらの変数が、現在の時刻までの系列の履歴情報を保持していることがわかる。これは、ニューラルネットワークの現在の時刻における状態、あるいは記憶のようなものである。したがって、このような隠れ層出力は 隠れ状態と呼ばれる。隠れ状態は、前の時刻の定義を現在の時刻でそのまま用いるため、(12.4.5) の計算は 再帰的である。したがって、先ほど述べたように、再帰的計算に基づく隠れ状態を持つニューラルネットワークは 再帰型ニューラルネットワークと呼ばれる。RNN において (12.4.5) の計算を行う層は、再帰層と呼ばれる。

RNN の構成方法にはさまざまなものがある。(12.4.5) で定義される隠れ状態を持つものは非常に一般的である。時刻 t に対して、出力層の出力は MLP の計算と同様である。

$$\mathbf{O}_t = \mathbf{H}_t\mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}. \tag{12.4.6}$$

RNN のパラメータには、隠れ層の重み $\mathbf{W}_{\text{xh}} \in \mathbb{R}^{d\times h}, \mathbf{W}_{\text{hh}} \in \mathbb{R}^{h\times h}$、バイアス $\mathbf{b}_{\text{h}} \in \mathbb{R}^{1\times h}$、および出力層の重み $\mathbf{W}_{\text{hq}} \in \mathbb{R}^{h\times q}$ とバイアス $\mathbf{b}_{\text{q}} \in \mathbb{R}^{1\times q}$ が含まれる。特筆すべき点として、異なる時刻であっても、RNN は常にこれらのモデルパラメータを使用する。したがって、RNN のパラメータ化コストは時刻数が増えても増加しない。

Fig. 12.4.1 は、3 つの隣接する時刻における RNN の計算ロジックを示している。任意の時刻 t において、隠れ状態の計算は次のように扱える。(i) 現在の時刻 t の入力 $\mathbf{X}_t$ と前の時刻 $t-1$ の隠れ状態 $\mathbf{H}_{t-1}$ を連結する。(ii) その連結結果を活性化関数 ϕ を持つ全結合層に入力する。このような全結合層の出力が、現在の時刻 t の隠れ状態 $\mathbf{H}_t$ である。この場合、モデルパラメータは (12.4.5) にある $\mathbf{W}_{\text{xh}}$ と $\mathbf{W}_{\text{hh}}$ の連結、およびバイアス $\mathbf{b}_{\text{h}}$ である。現在の時刻 t の隠れ状態 $\mathbf{H}_t$ は、次

の時刻 $t+1$ の隠れ状態 $\mathbf{H}_{t+1}$ の計算に関与する。さらに、$\mathbf{H}_t$ は全結合の出力層にも入力され、現在の時刻 t の出力 $\mathbf{O}_t$ を計算する。

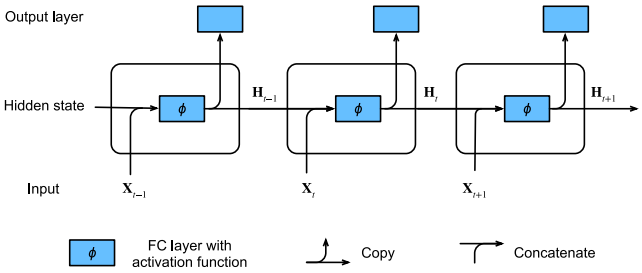


Fig. 12.4.1 隠れ状態を持つ RNN。

先ほど、隠れ状態のための $\mathbf{X}_t\mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hh}}$ の計算は、$\mathbf{X}_t$ と $\mathbf{H}_{t-1}$ の連結と、$\mathbf{W}_{\text{xh}}$ と $\mathbf{W}_{\text{hh}}$ の連結との行列積に等しいと述べた。これは数学的に証明できるが、以下では簡単なコード片で示すだけにする。まず、形状がそれぞれ (3, 1), (1, 4), (3, 4), (4, 4) の行列 `X`, `W_xh`, `H`, `W_hh` を定義する。`X` に `W_xh` を掛け、`H` に `W_hh` を掛け、その 2 つの積を加えると、形状 (3, 4) の行列が得られる。

```
X, W_xh = d2l.randn(3, 1), d2l.randn(1, 4)
H, W_hh = d2l.randn(3, 4), d2l.randn(4, 4)
d2l.matmul(X, W_xh) + d2l.matmul(H, W_hh)
```

```
X, W_xh = d2l.randn(3, 1), d2l.randn(1, 4)
H, W_hh = d2l.randn(3, 4), d2l.randn(4, 4)
d2l.matmul(X, W_xh) + d2l.matmul(H, W_hh)
```

```
X, W_xh = jax.random.normal(d2l.get_key(), (3, 1)), jax.random.normal(
                                                        d2l.get_key(), (1, 4))
H, W_hh = jax.random.normal(d2l.get_key(), (3, 4)), jax.random.normal(
                                                        d2l.get_key(), (4, 4))
d2l.matmul(X, W_xh) + d2l.matmul(H, W_hh)
```

```
X, W_xh = d2l.normal((3, 1)), d2l.normal((1, 4))
H, W_hh = d2l.normal((3, 4)), d2l.normal((4, 4))
d2l.matmul(X, W_xh) + d2l.matmul(H, W_hh)
```

次に、行方向（axis 1）に沿って行列 `X` と `H` を連結し、列方向（axis 0）に沿って行列 `W_xh` と `W_hh` を連結する。これら 2 つの連結結果は、それぞれ形状 (3, 5) と (5, 4) の行列になる。この 2 つの連結した行列を掛け合わせると、上と同じ形状 (3, 4) の出力行列が得られる。

```
d2l.matmul(d2l.concat((X, H), 1), d2l.concat((W_xh, W_hh), 0))
```

12.4.3 RNN に基づく文字レベル言語モデル

Section 12.3 での言語モデリングでは、現在および過去のトークンに基づいて次のトークンを予測することを目指した。そのため、元の系列を 1 トークンずらしたものを目標（ラベル）として用いる。Bengio *et al.* (2003) は、言語モデリングにニューラルネットワークを用いることを最初に提案した。以下では、RNN を用いて言語モデルを構築する方法を示す。ミニバッチサイズを 1 とし、テキスト系列を“machine”とする。後続の節での学習を簡単にするため、テキストを単語ではなく文字にトークン化し、文字レベル言語モデルを考える。Fig. 12.4.2 は、文字レベル言語モデリングのために RNN を通じて現在および過去の文字に基づいて次の文字を予測する方法を示している。

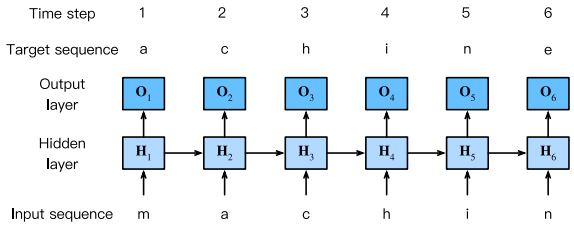


Fig. 12.4.2 RNN に基づく文字レベル言語モデル。入力系列と目標系列はそれぞれ“machin”と“achine”である。

学習過程では、各時刻の出力層からの出力に対して softmax 演算を行い、その後、交差エントロピー損失を用いてモデル出力と目標との誤差を計算する。隠れ層における隠れ状態の再帰的計算のため、Fig. 12.4.2 の時刻 3 における出力 $\mathbf{O}_3$ は、テキスト系列“m”, “a”, “c”によって決まる。系列の次の文字は学習データ中では“h”なので、時刻 3 の損失は、特徴系列“m”, “a”, “c”に基づいて生成された次の文字の確率分布と、この時刻の目標“h”に依存する。

実際には、各トークンは d 次元ベクトルで表され、バッチサイズ $n > 1$ を用いる。したがって、時刻 t における入力 $\mathbf{X}_t$ は $n \times d$ 行列となり、これは Section 12.4.2 で説明した内容と同じである。

以下の節では、文字レベル言語モデルのための RNN を実装する。

12.4.4 まとめ

隠れ状態に対して再帰的計算を用いるニューラルネットワークを、再帰型ニューラルネットワーク（RNN）と呼ぶ。RNN の隠れ状態は、現在の時刻までの系列の履歴情報を捉えることができる。再帰的計算を用いることで、RNN のモデルパラメータ数は時刻数が増えても増加しない。応用として、RNN は文字レベル言語モデルの構築に利用できる。

12.4.5 演習

1. RNN を用いてテキスト系列中の次の文字を予測する場合、任意の出力に必要な次元はどれくらいか。

2. なぜ RNN は、テキスト系列中のある時刻におけるトークンの条件付き確率を、それ以前のすべてのトークンに基づいて表現できるのか。
3. 長い系列を逆伝播すると、勾配はどうなるか。
4. この節で説明した言語モデルに関連する問題にはどのようなものがあるか。

12.5 ゼロからの再帰ニューラルネットワークの実装

これで、RNN をゼロから実装する準備が整った。特に、この RNN を文字レベルの言語モデルとして機能するように学習させる (Section 12.4 を参照)。また、H. G. ウェルズの *The Time Machine* の全文からなるコーパスで学習し、Section 12.2 で概説したデータ処理手順に従う。まずデータセットを読み込む。

```
%load_ext d2lbook.tab
tab.interact_select('mxnet', 'pytorch', 'tensorflow', 'jax')
```

```
%matplotlib inline
from d2l import torch as d2l
import math
import torch
from torch import nn
from torch.nn import functional as F
```

```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import autograd, gluon, np, npx
npx.set_np()
```

```
%matplotlib inline
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
import math
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import math
import tensorflow as tf
```

12.5.1 RNN モデル

まず、RNN モデルを実装するためのクラスを定義する (Section 12.4.2)。隠れユニット数 num_hiddens は調整可能なハイパーパラメータであることに注意せよ。

```
class RNNScratch(d2l.Module):  #@save
    """The RNN model implemented from scratch."""
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.W_xh = d2l.randn(num_inputs, num_hiddens) * sigma
            self.W_hh = d2l.randn(
                num_hiddens, num_hiddens) * sigma
            self.b_h = d2l.zeros(num_hiddens)
        if tab.selected('pytorch'):
            self.W_xh = nn.Parameter(
                d2l.randn(num_inputs, num_hiddens) * sigma)
            self.W_hh = nn.Parameter(
                d2l.randn(num_hiddens, num_hiddens) * sigma)
            self.b_h = nn.Parameter(d2l.zeros(num_hiddens))
        if tab.selected('tensorflow'):
            self.W_xh = tf.Variable(d2l.normal(
                (num_inputs, num_hiddens)) * sigma)
            self.W_hh = tf.Variable(d2l.normal(
                (num_hiddens, num_hiddens)) * sigma)
            self.b_h = tf.Variable(d2l.zeros(num_hiddens))
```

```
class RNNScratch(d2l.Module):  #@save
    """The RNN model implemented from scratch."""
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.W_xh = d2l.randn(num_inputs, num_hiddens) * sigma
            self.W_hh = d2l.randn(
                num_hiddens, num_hiddens) * sigma
            self.b_h = d2l.zeros(num_hiddens)
        if tab.selected('pytorch'):
            self.W_xh = nn.Parameter(
                d2l.randn(num_inputs, num_hiddens) * sigma)
            self.W_hh = nn.Parameter(
                d2l.randn(num_hiddens, num_hiddens) * sigma)
            self.b_h = nn.Parameter(d2l.zeros(num_hiddens))
        if tab.selected('tensorflow'):
            self.W_xh = tf.Variable(d2l.normal(
                (num_inputs, num_hiddens)) * sigma)
            self.W_hh = tf.Variable(d2l.normal(
                (num_hiddens, num_hiddens)) * sigma)
            self.b_h = tf.Variable(d2l.zeros(num_hiddens))
```

```
class RNNScratch(nn.Module):  #@save
    """The RNN model implemented from scratch."""
    num_inputs: int
```

(continues on next page)

(continued from previous page)

```
    num_hiddens: int
    sigma: float = 0.01

    def setup(self):
        self.W_xh = self.param('W_xh', nn.initializers.normal(self.sigma),
                               (self.num_inputs, self.num_hiddens))
        self.W_hh = self.param('W_hh', nn.initializers.normal(self.sigma),
                               (self.num_hiddens, self.num_hiddens))
        self.b_h = self.param('b_h', nn.initializers.zeros, (self.num_hiddens))
```

```
class RNNScratch(d2l.Module):  #@save
    """The RNN model implemented from scratch."""
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.W_xh = d2l.randn(num_inputs, num_hiddens) * sigma
            self.W_hh = d2l.randn(
                num_hiddens, num_hiddens) * sigma
            self.b_h = d2l.zeros(num_hiddens)
        if tab.selected('pytorch'):
            self.W_xh = nn.Parameter(
                d2l.randn(num_inputs, num_hiddens) * sigma)
            self.W_hh = nn.Parameter(
                d2l.randn(num_hiddens, num_hiddens) * sigma)
            self.b_h = nn.Parameter(d2l.zeros(num_hiddens))
        if tab.selected('tensorflow'):
            self.W_xh = tf.Variable(d2l.normal(
                (num_inputs, num_hiddens)) * sigma)
            self.W_hh = tf.Variable(d2l.normal(
                (num_hiddens, num_hiddens)) * sigma)
            self.b_h = tf.Variable(d2l.zeros(num_hiddens))
```

以下の `forward` メソッドは、現在の入力と前の時刻のモデル状態が与えられたときに、任意の時刻における出力と隠れ状態をどのように計算するかを定義する。RNN モデルは `inputs` の最外側の次元に沿ってループし、隠れ状態を 1 時刻ずつ更新することに注意せよ。ここでのモデルは tanh 活性化関数を使う (Section 8.1.2)。

```
@d2l.add_to_class(RNNScratch)  #@save
def forward(self, inputs, state=None):
    if state is None:
        # Initial state with shape: (batch_size, num_hiddens)
        if tab.selected('mxnet'):
            state = d2l.zeros((inputs.shape[1], self.num_hiddens),
                              ctx=inputs.ctx)
        if tab.selected('pytorch'):
            state = d2l.zeros((inputs.shape[1], self.num_hiddens),
                              device=inputs.device)
        if tab.selected('tensorflow'):
            state = d2l.zeros((inputs.shape[1], self.num_hiddens))
    else:
```

(continues on next page)

(continued from previous page)

```
        state, = state
        if tab.selected('tensorflow'):
            state = d2l.reshape(state, (-1, self.num_hiddens))
    outputs = []
    for X in inputs:  # Shape of inputs: (num_steps, batch_size, num_inputs)
        state = d2l.tanh(d2l.matmul(X, self.W_xh) +
                         d2l.matmul(state, self.W_hh) + self.b_h)
        outputs.append(state)
    return outputs, state
```

```
@d2l.add_to_class(RNNScratch)  #@save
def forward(self, inputs, state=None):
    if state is None:
        # Initial state with shape: (batch_size, num_hiddens)
        if tab.selected('mxnet'):
            state = d2l.zeros((inputs.shape[1], self.num_hiddens),
                              ctx=inputs.ctx)
        if tab.selected('pytorch'):
            state = d2l.zeros((inputs.shape[1], self.num_hiddens),
                              device=inputs.device)
        if tab.selected('tensorflow'):
            state = d2l.zeros((inputs.shape[1], self.num_hiddens))
    else:
        state, = state
        if tab.selected('tensorflow'):
            state = d2l.reshape(state, (-1, self.num_hiddens))
    outputs = []
    for X in inputs:  # Shape of inputs: (num_steps, batch_size, num_inputs)
        state = d2l.tanh(d2l.matmul(X, self.W_xh) +
                         d2l.matmul(state, self.W_hh) + self.b_h)
        outputs.append(state)
    return outputs, state
```

```
@d2l.add_to_class(RNNScratch)  #@save
def __call__(self, inputs, state=None):
    if state is not None:
        state, = state
    outputs = []
    for X in inputs:  # Shape of inputs: (num_steps, batch_size, num_inputs)
        state = d2l.tanh(d2l.matmul(X, self.W_xh) + (
            d2l.matmul(state, self.W_hh) if state is not None else 0)
                         + self.b_h)
        outputs.append(state)
    return outputs, state
```

```
@d2l.add_to_class(RNNScratch)  #@save
def forward(self, inputs, state=None):
    if state is None:
        # Initial state with shape: (batch_size, num_hiddens)
        if tab.selected('mxnet'):
            state = d2l.zeros((inputs.shape[1], self.num_hiddens),
                              ctx=inputs.ctx)
```

(continues on next page)

(continued from previous page)

```
        if tab.selected('pytorch'):
            state = d2l.zeros((inputs.shape[1], self.num_hiddens),
                              device=inputs.device)
        if tab.selected('tensorflow'):
            state = d2l.zeros((inputs.shape[1], self.num_hiddens))
    else:
        state, = state
        if tab.selected('tensorflow'):
            state = d2l.reshape(state, (-1, self.num_hiddens))
    outputs = []
    for X in inputs:  # Shape of inputs: (num_steps, batch_size, num_inputs)
        state = d2l.tanh(d2l.matmul(X, self.W_xh) +
                         d2l.matmul(state, self.W_hh) + self.b_h)
        outputs.append(state)
    return outputs, state
```

RNN モデルに入力シーケンスのミニバッチを次のように与えることができる。

```
batch_size, num_inputs, num_hiddens, num_steps = 2, 16, 32, 100
rnn = RNNScratch(num_inputs, num_hiddens)
X = d2l.ones((num_steps, batch_size, num_inputs))
outputs, state = rnn(X)
```

```
batch_size, num_inputs, num_hiddens, num_steps = 2, 16, 32, 100
rnn = RNNScratch(num_inputs, num_hiddens)
X = d2l.ones((num_steps, batch_size, num_inputs))
outputs, state = rnn(X)
```

```
batch_size, num_inputs, num_hiddens, num_steps = 2, 16, 32, 100
rnn = RNNScratch(num_inputs, num_hiddens)
X = d2l.ones((num_steps, batch_size, num_inputs))
(outputs, state), _ = rnn.init_with_output(d2l.get_key(), X)
```

```
batch_size, num_inputs, num_hiddens, num_steps = 2, 16, 32, 100
rnn = RNNScratch(num_inputs, num_hiddens)
X = d2l.ones((num_steps, batch_size, num_inputs))
outputs, state = rnn(X)
```

RNN モデルが正しい形状の結果を出力し、隠れ状態の次元が変わらないことを確認しよう。

```
def check_len(a, n):  #@save
    """Check the length of a list."""
    assert len(a) == n, f'list\'s length {len(a)} != expected length {n}'

def check_shape(a, shape):  #@save
    """Check the shape of a tensor."""
    assert a.shape == shape, \
```

(continues on next page)

(continued from previous page)

```
            f'tensor\'s shape {a.shape} != expected shape {shape}'

check_len(outputs, num_steps)
check_shape(outputs[0], (batch_size, num_hiddens))
check_shape(state, (batch_size, num_hiddens))
```

12.5.2 RNN ベースの言語モデル

以下の RNNLMScratch クラスは、RNN ベースの言語モデルを定義する。ここでは __init__ メソッドの rnn 引数を通して RNN を渡す。言語モデルを学習するとき、入力と出力は同じ語彙から来る。したがって、それらの次元は同じであり、語彙サイズに等しくなる。モデルの評価には困惑度を使うことに注意せよ。Section 12.3.2 で述べたように、これにより長さの異なる系列を比較できる。

```
class RNNLMScratch(d2l.Classifier):  #@save
    """The RNN-based language model implemented from scratch."""
    def __init__(self, rnn, vocab_size, lr=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.init_params()

    def init_params(self):
        self.W_hq = nn.Parameter(
            d2l.randn(
                self.rnn.num_hiddens, self.vocab_size) * self.rnn.sigma)
        self.b_q = nn.Parameter(d2l.zeros(self.vocab_size))

    def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('ppl', d2l.exp(l), train=True)
        return l

    def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('ppl', d2l.exp(l), train=False)
```

```
class RNNLMScratch(d2l.Classifier):  #@save
    """The RNN-based language model implemented from scratch."""
    def __init__(self, rnn, vocab_size, lr=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.init_params()

    def init_params(self):
        if tab.selected('mxnet'):
            self.W_hq = d2l.randn(
                self.rnn.num_hiddens, self.vocab_size) * self.rnn.sigma
            self.b_q = d2l.zeros(self.vocab_size)
            for param in self.get_scratch_params():
```

(continues on next page)

(continued from previous page)

```
                param.attach_grad()
        if tab.selected('tensorflow'):
            self.W_hq = tf.Variable(d2l.normal(
                (self.rnn.num_hiddens, self.vocab_size)) * self.rnn.sigma)
            self.b_q = tf.Variable(d2l.zeros(self.vocab_size))

    def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('ppl', d2l.exp(l), train=True)
        return l

    def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('ppl', d2l.exp(l), train=False)
```

```
class RNNLMScratch(d2l.Classifier):  #@save
    """The RNN-based language model implemented from scratch."""
    rnn: nn.Module
    vocab_size: int
    lr: float = 0.01

    def setup(self):
        self.W_hq = self.param('W_hq', nn.initializers.normal(self.rnn.sigma),
                               (self.rnn.num_hiddens, self.vocab_size))
        self.b_q = self.param('b_q', nn.initializers.zeros, (self.vocab_size))

    def training_step(self, params, batch, state):
        value, grads = jax.value_and_grad(
            self.loss, has_aux=True)(params, batch[:-1], batch[-1], state)
        l, _ = value
        self.plot('ppl', d2l.exp(l), train=True)
        return value, grads

    def validation_step(self, params, batch, state):
        l, _ = self.loss(params, batch[:-1], batch[-1], state)
        self.plot('ppl', d2l.exp(l), train=False)
```

```
class RNNLMScratch(d2l.Classifier):  #@save
    """The RNN-based language model implemented from scratch."""
    def __init__(self, rnn, vocab_size, lr=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.init_params()

    def init_params(self):
        if tab.selected('mxnet'):
            self.W_hq = d2l.randn(
                self.rnn.num_hiddens, self.vocab_size) * self.rnn.sigma
            self.b_q = d2l.zeros(self.vocab_size)
            for param in self.get_scratch_params():
                param.attach_grad()
        if tab.selected('tensorflow'):
            self.W_hq = tf.Variable(d2l.normal(
```

(continues on next page)

(continued from previous page)

```
                (self.rnn.num_hiddens, self.vocab_size)) * self.rnn.sigma)
            self.b_q = tf.Variable(d2l.zeros(self.vocab_size))

    def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('ppl', d2l.exp(l), train=True)
        return l

    def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('ppl', d2l.exp(l), train=False)
```

ワンホットエンコーディング

各トークンは、対応する単語/文字/単語片が語彙内のどの位置にあるかを示す数値インデックスで表されることを思い出してほしい。各時刻に 1 つの入力ノードだけを持つニューラルネットワークを構築し、そのインデックスをスカラー値として入力することを考えたくなるかもしれない。これは、価格や温度のような数値入力を扱う場合にはうまくいく。そのような場合、十分に近い任意の 2 つの値は同様に扱うべきだからである。しかし、これはここではあまり意味をなさない。語彙の 45 番目と 46 番目の単語はたまたま“their”と“said”であるが、その意味はまったく似ていない。

このようなカテゴリデータを扱うとき、最も一般的な戦略は各項目を ワンホットエンコーディングで表すことである (Section 7.1.1 を参照)。ワンホットエンコーディングとは、長さが語彙サイズ N で与えられるベクトルであり、トークンに対応する要素だけが 1 に設定され、それ以外の要素はすべて 0 に設定される。たとえば、語彙に 5 個の要素があるなら、インデックス 0 と 2 に対応するワンホットベクトルは次のようになる。

```
F.one_hot(torch.tensor([0, 2]), 5)
```

```
tensor([[1, 0, 0, 0, 0],
        [0, 0, 1, 0, 0]])
```

```
npx.one_hot(np.array([0, 2]), 5)
```

```
array([[1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0.]])
```

```
jax.nn.one_hot(jnp.array([0, 2]), 5)
```

```
Array([[1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0.]], dtype=float32)
```

```
tf.one_hot(tf.constant([0, 2]), 5)
```

```
<tf.Tensor: shape=(2, 5), dtype=float32, numpy=
array([[1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0.]], dtype=float32)>
```

各反復でサンプリングされるミニバッチは、(バッチサイズ, 時間ステップ数) の形を取る。各入力をワンホットベクトルとして表現すると、各ミニバッチは 3 次元テンソルとみなせる。その第 3 軸方向の長さは語彙サイズ (len(vocab)) で与えられる。入力を転置して、(時間ステップ数, バッチサイズ, 語彙サイズ) の形の出力を得ることがよくある。これにより、ミニバッチの隠れ状態を時刻ごとに更新するために、最外側の次元に沿ってより便利にループできる（たとえば、上の forward メソッドのように）。

```
@d2l.add_to_class(RNNLMScratch)  #@save
def one_hot(self, X):
    # Output shape: (num_steps, batch_size, vocab_size)
    if tab.selected('mxnet'):
        return npx.one_hot(X.T, self.vocab_size)
    if tab.selected('pytorch'):
        return F.one_hot(X.T, self.vocab_size).type(torch.float32)
    if tab.selected('tensorflow'):
        return tf.one_hot(tf.transpose(X), self.vocab_size)
    if tab.selected('jax'):
        return jax.nn.one_hot(X.T, self.vocab_size)
```

RNN 出力の変換

言語モデルは全結合出力層を使って、各時刻の RNN 出力をトークン予測へ変換する。

```
@d2l.add_to_class(RNNLMScratch)  #@save
def output_layer(self, rnn_outputs):
    outputs = [d2l.matmul(H, self.W_hq) + self.b_q for H in rnn_outputs]
    return d2l.stack(outputs, 1)

@d2l.add_to_class(RNNLMScratch)  #@save
def forward(self, X, state=None):
    embs = self.one_hot(X)
    rnn_outputs, _ = self.rnn(embs, state)
    return self.output_layer(rnn_outputs)
```

順伝播計算が正しい形状の出力を生成するか確認しよう。

```
model = RNNLMScratch(rnn, num_inputs)
outputs = model(d2l.ones((batch_size, num_steps), dtype=d2l.int64))
check_shape(outputs, (batch_size, num_steps, num_inputs))
```

```
model = RNNLMScratch(rnn, num_inputs)
outputs = model(d2l.ones((batch_size, num_steps), dtype=d2l.int64))
check_shape(outputs, (batch_size, num_steps, num_inputs))
```

```
model = RNNLMScratch(rnn, num_inputs)
outputs, _ = model.init_with_output(d2l.get_key(),
                                    d2l.ones((batch_size, num_steps),
                                             dtype=d2l.int32))
check_shape(outputs, (batch_size, num_steps, num_inputs))
```

```
model = RNNLMScratch(rnn, num_inputs)
outputs = model(d2l.ones((batch_size, num_steps), dtype=d2l.int64))
check_shape(outputs, (batch_size, num_steps, num_inputs))
```

12.5.3 勾配クリッピング

すでに、ニューラルネットワークを「深い」と考えるとき、単一の時刻の中でも入力と出力の間に多くの層があるという意味で深いことに慣れているだろうが、系列の長さは新たな深さの概念を導入する。入力から出力への方向でネットワークを通過することに加えて、最初の時刻の入力は、モデルの最終時刻での出力に影響を与えるために、時刻に沿って T 層からなる連鎖を通過しなければならない。逆方向から見ると、各反復で時間方向に逆伝播を行うため、長さ $O(T)$ の行列積の連鎖が生じる。Section 8.4 で述べたように、これは数値的不安定性を引き起こし、重み行列の性質に応じて勾配が爆発したり消失したりする。

消失勾配と爆発勾配への対処は、RNN を設計するうえでの根本的な問題であり、現代のニューラルネットワークアーキテクチャにおける最大級の進歩のいくつかを生み出す原動力となってきた。次の章では、消失勾配問題の緩和を目指して設計された特殊なアーキテクチャについて説明する。しかし、現代の RNN であっても、爆発勾配に悩まされることは少なくない。洗練されてはいないものの広く使われている解決策の 1 つは、勾配を単純にクリップして、結果として得られる「クリップされた」勾配の値を小さくすることである。

一般に、勾配降下法で何らかの目的関数を最適化するとき、たいていベクトルのようなパラメータ $\mathbf{x}$ を反復的に更新するが、その際には負の勾配 $\mathbf{g}$ の方向へ押し進める（確率的勾配降下法では、この勾配をランダムにサンプリングしたミニバッチ上で計算する）。たとえば、学習率 $\eta > 0$ のとき、各更新は $\mathbf{x} \leftarrow \mathbf{x} - \eta\mathbf{g}$ の形を取る。さらに、目的関数 f が十分に滑らかであると仮定しよう。形式的には、目的関数が定数 L の *Lipschitz* 連続であるといい、任意の $\mathbf{x}$ と $\mathbf{y}$ に対して

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L\|\mathbf{x} - \mathbf{y}\|. \tag{12.5.1}$$

ご覧のとおり、パラメータベクトルから $\eta\mathbf{g}$ を引いて更新するとき、目的関数の

値の変化は、学習率、勾配のノルム、および L に次のように依存する。

$$|f(\mathbf{x}) - f(\mathbf{x} - \eta\mathbf{g})| \leq L\eta\|\mathbf{g}\|. \tag{12.5.2}$$

言い換えると、目的関数は $L\eta\|\mathbf{g}\|$ を超えて変化することはない。この上界が小さいことは、良いことにも悪いことにもなりえる。欠点としては、目的関数の値を減らせる速度を制限してしまうことである。一方で、1 回の勾配ステップでどれだけ大きく失敗しうるかを抑えるという利点がある。

勾配が爆発するとは、$\|\mathbf{g}\|$ が過度に大きくなることを意味する。この最悪の場合、1 回の勾配ステップで非常に大きな損害を与えてしまい、数千回の学習反復で得られた進歩をすべて打ち消してしまうことさえある。勾配が非常に大きくなりうると、ニューラルネットワークの学習はしばしば発散し、目的関数の値を減らせなくなる。また別の場合には、最終的には収束するものの、損失の大きなスパイクのために不安定になる。

$L\eta\|\mathbf{g}\|$ の大きさを抑える 1 つの方法は、学習率 η を非常に小さな値に縮小することである。この利点は、更新にバイアスを導入しないことである。しかし、大きな勾配が起こるのが まれでしかない場合はどうだろうか。この思い切った対策は、まれな爆発勾配イベントに対処するためだけに、すべてのステップでの進歩を遅くしてしまう。よく使われる代替案は、勾配クリッピングのヒューリスティックを採用し、次のように勾配 $\mathbf{g}$ を半径 θ の球へ射影することである。

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}. \tag{12.5.3}$$

これにより、勾配ノルムが θ を超えないことが保証され、更新後の勾配が元の $\mathbf{g}$ の方向と完全に整列したままであることも保証される。また、任意のミニバッチ（およびその中の任意のサンプル）がパラメータベクトルに及ぼしうる影響を制限するという望ましい副作用もある。これにより、モデルにある程度の頑健性が与えられる。はっきり言えば、これはハックである。勾配クリッピングは、常に真の勾配に従っているわけではないことを意味し、考えられる副作用を解析的に理解するのは困難である。しかし、非常に有用なハックであり、ほとんどの深層学習フレームワークにおける RNN 実装で広く採用されている。

以下では、勾配をクリップするメソッドを定義する。これは `d2l.Trainer` クラスの `fit_epoch` メソッドから呼び出される (Section 6.4 を参照)。勾配ノルムを計算するときは、すべてのモデルパラメータを連結して、1 つの巨大なパラメータベクトルとして扱っていることに注意せよ。

```
@d2l.add_to_class(d2l.Trainer)  #@save
def clip_gradients(self, grad_clip_val, model):
    params = [p for p in model.parameters() if p.requires_grad]
    norm = torch.sqrt(sum(torch.sum((p.grad ** 2)) for p in params))
    if norm > grad_clip_val:
        for param in params:
            param.grad[:] *= grad_clip_val / norm
```

```
@d2l.add_to_class(d2l.Trainer)  #@save
def clip_gradients(self, grad_clip_val, model):
    params = model.parameters()
    if not isinstance(params, list):
        params = [p.data() for p in params.values()]
    norm = math.sqrt(sum((p.grad ** 2).sum() for p in params))
    if norm > grad_clip_val:
        for param in params:
            param.grad[:] *= grad_clip_val / norm
```

```
@d2l.add_to_class(d2l.Trainer)  #@save
def clip_gradients(self, grad_clip_val, grads):
    grad_leaves, _ = jax.tree_util.tree_flatten(grads)
    norm = jnp.sqrt(sum(jnp.vdot(x, x) for x in grad_leaves))
    clip = lambda grad: jnp.where(norm < grad_clip_val,
                                  grad, grad * (grad_clip_val / norm))
    return jax.tree_util.tree_map(clip, grads)
```

```
@d2l.add_to_class(d2l.Trainer)  #@save
def clip_gradients(self, grad_clip_val, grads):
    grad_clip_val = tf.constant(grad_clip_val, dtype=tf.float32)
    new_grads = [tf.convert_to_tensor(grad) if isinstance(
        grad, tf.IndexedSlices) else grad for grad in grads]
    norm = tf.math.sqrt(sum((tf.reduce_sum(grad ** 2)) for grad in new_grads))
    if tf.greater(norm, grad_clip_val):
        for i, grad in enumerate(new_grads):
            new_grads[i] = grad * grad_clip_val / norm
        return new_grads
    return grads
```

12.5.4 学習

The Time Machine データセット (data) を使って、ゼロから実装した RNN (rnn) に基づく文字レベルの言語モデル (model) を学習する。まず勾配を計算し、次にそれらをクリップし、最後にクリップされた勾配を使ってモデルパラメータを更新することに注意せよ。

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
if tab.selected('mxnet', 'pytorch', 'jax'):
    rnn = RNNScratch(num_inputs=len(data.vocab), num_hiddens=32)
    model = RNNLMScratch(rnn, vocab_size=len(data.vocab), lr=1)
    trainer = d2l.Trainer(max_epochs=100, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        rnn = RNNScratch(num_inputs=len(data.vocab), num_hiddens=32)
        model = RNNLMScratch(rnn, vocab_size=len(data.vocab), lr=1)
    trainer = d2l.Trainer(max_epochs=100, gradient_clip_val=1)
trainer.fit(model, data)
```

12.5.5 デコード

言語モデルが学習されると、次のトークンを予測するだけでなく、それ以降の各トークンを順に予測し続けることができる。その際、直前に予測したトークンを次の入力であるかのように扱う。文書の先頭から始めるかのようにテキストを生成したいだけの場合もある。しかし、多くの場合は、ユーザーが与えたプレフィックスに条件づけて言語モデルを使うと便利である。たとえば、検索エンジンのオートコンプリート機能を開発していたり、メール作成を支援したりする場合には、ユーザーがここまで入力した内容（プレフィックス）を与え、その続きとしてありそうなテキストを生成したいだろう。

以下の `predict` メソッドは、ユーザーが与えた `prefix` を取り込んだあと、1 文字ずつ継続を生成する。`prefix` の文字をループするとき、隠れ状態を次の時刻へ渡し続けるが、出力は生成しない。これを ウォームアップ期間と呼ぶ。プレフィックスを取り込んだ後は、以降の文字の出力を開始する準備が整う。それぞれの文字は次の時刻の入力としてモデルにフィードバックされる。

```
@d2l.add_to_class(RNNLMScratch)  #@save
def predict(self, prefix, num_preds, vocab, device=None):
    state, outputs = None, [vocab[prefix[0]]]
    for i in range(len(prefix) + num_preds - 1):
        if tab.selected('mxnet'):
            X = d2l.tensor([[outputs[-1]]], ctx=device)
        if tab.selected('pytorch'):
            X = d2l.tensor([[outputs[-1]]], device=device)
        if tab.selected('tensorflow'):
            X = d2l.tensor([[outputs[-1]]])
        embs = self.one_hot(X)
        rnn_outputs, state = self.rnn(embs, state)
        if i < len(prefix) - 1:  # Warm-up period
            outputs.append(vocab[prefix[i + 1]])
        else:  # Predict num_preds steps
            Y = self.output_layer(rnn_outputs)
            outputs.append(int(d2l.reshape(d2l.argmax(Y, axis=2), 1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```


```
@d2l.add_to_class(RNNLMScratch)  #@save
def predict(self, prefix, num_preds, vocab, device=None):
    state, outputs = None, [vocab[prefix[0]]]
    for i in range(len(prefix) + num_preds - 1):
        if tab.selected('mxnet'):
            X = d2l.tensor([[outputs[-1]]], ctx=device)
        if tab.selected('pytorch'):
            X = d2l.tensor([[outputs[-1]]], device=device)
        if tab.selected('tensorflow'):
            X = d2l.tensor([[outputs[-1]]])
        embs = self.one_hot(X)
        rnn_outputs, state = self.rnn(embs, state)
        if i < len(prefix) - 1:  # Warm-up period
            outputs.append(vocab[prefix[i + 1]])
        else:  # Predict num_preds steps
            Y = self.output_layer(rnn_outputs)
            outputs.append(int(d2l.reshape(d2l.argmax(Y, axis=2), 1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

```
@d2l.add_to_class(RNNLMScratch)  #@save
def predict(self, prefix, num_preds, vocab, params):
    state, outputs = None, [vocab[prefix[0]]]
    for i in range(len(prefix) + num_preds - 1):
        X = d2l.tensor([[outputs[-1]]])
        embs = self.one_hot(X)
        rnn_outputs, state = self.rnn.apply({'params': params['rnn']},
                                            embs, state)
        if i < len(prefix) - 1:  # Warm-up period
            outputs.append(vocab[prefix[i + 1]])
        else:  # Predict num_preds steps
            Y = self.apply({'params': params}, rnn_outputs,
                           method=self.output_layer)
            outputs.append(int(d2l.reshape(d2l.argmax(Y, axis=2), 1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

```
@d2l.add_to_class(RNNLMScratch)  #@save
def predict(self, prefix, num_preds, vocab, device=None):
    state, outputs = None, [vocab[prefix[0]]]
    for i in range(len(prefix) + num_preds - 1):
        if tab.selected('mxnet'):
            X = d2l.tensor([[outputs[-1]]], ctx=device)
        if tab.selected('pytorch'):
            X = d2l.tensor([[outputs[-1]]], device=device)
        if tab.selected('tensorflow'):
            X = d2l.tensor([[outputs[-1]]])
        embs = self.one_hot(X)
        rnn_outputs, state = self.rnn(embs, state)
        if i < len(prefix) - 1:  # Warm-up period
            outputs.append(vocab[prefix[i + 1]])
        else:  # Predict num_preds steps
            Y = self.output_layer(rnn_outputs)
            outputs.append(int(d2l.reshape(d2l.argmax(Y, axis=2), 1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

以下では、プレフィックスを指定して 20 文字を追加生成させる。

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
'it has at a meracall the t'
```

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
'it has i have all in the t'
```

```
model.predict('it has', 20, data.vocab, trainer.state.params)
```

```
'it has of the time the tim'
```

```
model.predict('it has', 20, data.vocab)
```

```
'it has of the time time th'
```

上の RNN モデルをゼロから実装することは学習になるが、便利ではない。次の節では、標準的なアーキテクチャを使って RNN を簡単に構築し、高度に最適化されたライブラリ関数に頼ることで性能向上を得る方法を見る。

12.5.6 まとめ

ユーザーが与えたテキストのプレフィックスに続くテキストを生成するように、RNN ベースの言語モデルを学習できる。単純な RNN 言語モデルは、入力エンコーディング、RNN によるモデリング、出力生成から構成される。学習中、勾配クリッピングは爆発勾配の問題を緩和できるが、消失勾配の問題は解決しない。実験では、単純な RNN 言語モデルを実装し、文字レベルでトークン化したテキスト系列に対して勾配クリッピング付きで学習させた。プレフィックスを条件にすることで、言語モデルを使ってありそうな続きのテキストを生成でき、これはオートコンプリート機能など多くの応用で有用である。

12.5.7 演習

1. 実装した言語モデルは、*The Time Machine* の最初のトークンまでのすべての過去トークンに基づいて次のトークンを予測するか。
2. 予測に使われる履歴の長さを制御するハイパーパラメータはどれか。
3. ワンホットエンコーディングが、各対象に対して異なる埋め込みを選ぶことと等価であることを示せ。
4. ハイパーパラメータ（たとえば、エポック数、隠れユニット数、ミニバッチ内の時間ステップ数、学習率）を調整して困惑度を改善せよ。この単純なアーキテクチャのままで、どこまで下げられるか。
5. ワンホットエンコーディングを学習可能な埋め込みに置き換えよ。これにより性能は向上するか。
6. *The Time Machine* で学習したこの言語モデルが、H. G. ウェルズの他の本、たとえば *The War of the Worlds* に対してどの程度うまく機能するかを調べる実験を行え。
7. 別の実験として、このモデルの困惑度を他の著者による本で評価せよ。
8. 予測方法を修正して、最もありそうな次の文字を選ぶのではなく、サンプリングを使うようにせよ。

- 何が起こるだろうか。
- たとえば、$q(x_t \mid x_{t-1}, \ldots, x_1) \propto P(x_t \mid x_{t-1}, \ldots, x_1)^\alpha$ で $\alpha > 1$ としてサンプリングすることで、モデルをよりありそうな出力に偏らせよ。

9. この節のコードを勾配クリッピングなしで実行せよ。何が起こるだろうか。
10. この節で使った活性化関数を ReLU に置き換え、この節の実験を繰り返せ。まだ勾配クリッピングは必要だろうか。なぜか。

12.6 リカレントニューラルネットワークの簡潔な実装

ほとんどのスクラッチ実装と同様に、Section 12.5 は各コンポーネントがどのように動作するかを理解できるように設計されていました。しかし、日常的に RNN を使うときや本番コードを書くときには、実装時間（一般的なモデルや関数に対するライブラリコードを提供してくれるため）と計算時間（これらのライブラリ実装を徹底的に最適化してくれるため）の両方を削減できるライブラリに、より頼りたくなるだろう。この節では、深層学習フレームワークが提供する高水準 API を用いて、同じ言語モデルをより効率的に実装する方法を示す。まずはこれまでと同様に、タイムマシンデータセットを読み込む。

```
%load_ext d2lbook.tab
tab.interact_select('mxnet', 'pytorch', 'tensorflow', 'jax')
```

```
from d2l import torch as d2l
import torch
from torch import nn
from torch.nn import functional as F
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn, rnn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from jax import numpy as jnp
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

12.6.1 モデルの定義

以下のクラスを、高水準 API で実装された RNN を用いて定義する。

```
class RNN(d2l.Module):  #@save
    """ 高水準 API で実装された RNN モデル。"""
    def __init__(self, num_inputs, num_hiddens):
        super().__init__()
        self.save_hyperparameters()
        self.rnn = nn.RNN(num_inputs, num_hiddens)

    def forward(self, inputs, H=None):
        return self.rnn(inputs, H)
```

```
class RNN(d2l.Module):  #@save
    """ 高水準 API で実装された RNN モデル。"""
    def __init__(self, num_hiddens):
        super().__init__()
        self.save_hyperparameters()
        self.rnn = rnn.RNN(num_hiddens)

    def forward(self, inputs, H=None):
        if H is None:
            H, = self.rnn.begin_state(inputs.shape[1], ctx=inputs.ctx)
        outputs, (H, ) = self.rnn(inputs, (H, ))
        return outputs, H
```

```
class RNN(nn.Module):  #@save
    """ 高水準 API で実装された RNN モデル。"""
    num_hiddens: int

    @nn.compact
    def __call__(self, inputs, H=None):
        raise NotImplementedError
```

```
class RNN(d2l.Module):  #@save
    """ 高水準 API で実装された RNN モデル。"""
    def __init__(self, num_hiddens):
        super().__init__()
        self.save_hyperparameters()
        self.rnn = tf.keras.layers.SimpleRNN(
            num_hiddens, return_sequences=True, return_state=True,
            time_major=True)

    def forward(self, inputs, H=None):
        outputs, H = self.rnn(inputs, H)
        return outputs, H
```

Section 12.5 の `RNNLMScratch` クラスを継承して、次の `RNNLM` クラスは RNN ベースの完全な言語モデルを定義する。別個の全結合出力層を作成する必要があることに注意せよ。

```
class RNNLM(d2l.RNNLMScratch):  #@save
    """ 高水準 API で実装された RNN ベースの言語モデル。"""
    def init_params(self):
        self.linear = nn.LazyLinear(self.vocab_size)

    def output_layer(self, hiddens):
        return d2l.swapaxes(self.linear(hiddens), 0, 1)
```

```
class RNNLM(d2l.RNNLMScratch):  #@save
    """ 高水準 API で実装された RNN ベースの言語モデル。"""
    def init_params(self):
        if tab.selected('mxnet'):
            self.linear = nn.Dense(self.vocab_size, flatten=False)
            self.initialize()
        if tab.selected('tensorflow'):
            self.linear = tf.keras.layers.Dense(self.vocab_size)

    def output_layer(self, hiddens):
        if tab.selected('mxnet'):
            return d2l.swapaxes(self.linear(hiddens), 0, 1)
        if tab.selected('tensorflow'):
            return d2l.transpose(self.linear(hiddens), (1, 0, 2))
```

```
class RNNLM(d2l.RNNLMScratch):  #@save
    """ 高水準 API で実装された RNN ベースの言語モデル。"""
    training: bool = True

    def setup(self):
        self.linear = nn.Dense(self.vocab_size)

    def output_layer(self, hiddens):
        return d2l.swapaxes(self.linear(hiddens), 0, 1)

    def forward(self, X, state=None):
        embs = self.one_hot(X)
        rnn_outputs, _ = self.rnn(embs, state, self.training)
        return self.output_layer(rnn_outputs)
```

```
class RNNLM(d2l.RNNLMScratch):  #@save
    """ 高水準 API で実装された RNN ベースの言語モデル。"""
    def init_params(self):
        if tab.selected('mxnet'):
            self.linear = nn.Dense(self.vocab_size, flatten=False)
            self.initialize()
        if tab.selected('tensorflow'):
            self.linear = tf.keras.layers.Dense(self.vocab_size)

    def output_layer(self, hiddens):
        if tab.selected('mxnet'):
            return d2l.swapaxes(self.linear(hiddens), 0, 1)
        if tab.selected('tensorflow'):
            return d2l.transpose(self.linear(hiddens), (1, 0, 2))
```

12.6.2 学習と予測

モデルを学習する前に、ランダムな重みで初期化されたモデルを使って予測してみよう。ネットワークはまだ学習されていないので、意味をなさない予測を生成する。

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
if tab.selected('mxnet', 'tensorflow'):
    rnn = RNN(num_hiddens=32)
if tab.selected('pytorch'):
    rnn = RNN(num_inputs=len(data.vocab), num_hiddens=32)
model = RNNLM(rnn, vocab_size=len(data.vocab), lr=1)
model.predict('it has', 20, data.vocab)
```

次に、高水準 API を活用して、モデルを学習する。

```
if tab.selected('mxnet', 'pytorch'):
    trainer = d2l.Trainer(max_epochs=100, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        trainer = d2l.Trainer(max_epochs=100, gradient_clip_val=1)
trainer.fit(model, data)
```

Section 12.5 と比べると、このモデルは同程度の困惑度を達成するが、最適化された実装のおかげでより高速に動作する。これまでと同様に、指定した接頭辞文字列に続く予測トークンを生成できる。

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
'it has it and the pean the'
```

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
'it has a dimensions the ti'
```

```
model.predict('it has', 20, data.vocab)
```

```
'it has in the the the that'
```

12.6.3 まとめ

深層学習フレームワークの高水準 API は、標準的な RNN の実装を提供する。これらのライブラリを使えば、標準モデルを再実装するために時間を浪費せずに済む。さらに、フレームワークの実装はしばしば高度に最適化されているため、スクラッチ実装と比べて大幅な（計算）性能向上が得られる。

12.6.4 演習

1. 高水準 API を使って RNN モデルを過学習させることはできるだろうか。
2. Section 12.1 の自己回帰モデルを RNN を用いて実装せよ。

12.7 時間を通した逆伝播

Section 12.5 の演習を終えていれば、勾配クリッピングが、まれに発生する巨大な勾配が学習を不安定にするのを防ぐうえで不可欠であることを見てきたはずである。爆発する勾配は、長い系列にわたって逆伝播することに起因すると示唆した。現代的な RNN アーキテクチャを多数紹介する前に、逆伝播が系列モデルで数学的にどのように機能するのかを、もう少し詳しく見てみよう。この議論によって、消失勾配と 爆発勾配という概念にいくらか精密さが加わることを期待している。Section 8.3 で MLP を導入したときに、計算グラフを通した順伝播と逆伝播についての議論を思い出していただければ、RNN における順伝播は比較的わかりやすいはずである。RNN に逆伝播を適用することを 時間を通した逆伝播と呼ぶ (Werbos, 1990)。この手続きでは、RNN の計算グラフを 1 タイムステップずつ展開（またはアンロール）する必要がある。アンロールされた RNN は本質的にフィードフォワード型のニューラルネットワークであり、同じパラメータがアンロールされたネットワーク全体で繰り返し現れ、各タイムステップに登場するという特別な性質を持っている。その後は、通常のフィードフォワード型ニューラルネットワークと同様に、連鎖律を適用して、アンロールされたネットワークを通して勾配を逆伝播できる。各パラメータに関する勾配は、アンロールされたネットワーク内でそのパラメータが現れるすべての箇所にわたって和を取らなければならない。このような重み共有の扱いは、畳み込みニューラルネットワークの章ですでに見てきたはずである。

系列はかなり長くなりうるため、問題が生じる。1000 個を超えるトークンからなるテキスト系列を扱うことは珍しくない。これは、計算量の観点（メモリが多すぎる）と最適化の観点（数値的不安定性）の両方で問題を引き起こす。最初のステップからの入力は、出力に到達するまでに 1000 回以上の行列積を通過し、勾配を計算するためにもさらに 1000 回の行列積が必要である。ここでは、何がうまくいかなくなるのか、そして実際にはどう対処するのかを分析する。

12.7.1 RNN における勾配の解析

まず、RNN がどのように動作するかについての単純化したモデルから始める。このモデルでは、隠れ状態の具体的な詳細やその更新方法は無視する。ここでの数学記法では、スカラー、ベクトル、行列を明示的に区別しない。あくまで直感を養うことが目的である。この単純化したモデルでは、時刻 t における隠れ状態を h_t、入力を x_t、出力を o_t と表す。Section 12.4.2 で議論したように、入力

と隠れ状態は、隠れ層の 1 つの重み変数で掛け算される前に連結できる。したがって、隠れ層と出力層の重みをそれぞれ w_h と w_o で表す。その結果、各タイムステップにおける隠れ状態と出力は

$$\begin{aligned} h_t &= f(x_t, h_{t-1}, w_\mathrm{h}), \\ o_t &= g(h_t, w_\mathrm{o}), \end{aligned} \tag{12.7.1}$$

となる。ここで f と g はそれぞれ隠れ層と出力層の変換である。したがって、相互に依存する $\{\ldots, (x_{t-1}, h_{t-1}, o_{t-1}), (x_t, h_t, o_t), \ldots\}$ という値の連鎖が、再帰的な計算を通じて得られる。順伝播はかなり単純である。必要なのは、(x_t, h_t, o_t) の組を 1 タイムステップずつ順にループすることだけである。その後、出力 o_t と望ましい目標 y_t の差は、全 T タイムステップにわたる目的関数で評価され、

$$L(x_1, \ldots, x_T, y_1, \ldots, y_T, w_\mathrm{h}, w_\mathrm{o}) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t). \tag{12.7.2}$$

逆伝播では、特に目的関数 L のパラメータ w_h に関する勾配を計算するときに、少し厄介になる。具体的には、連鎖律により、

$$\begin{aligned} \frac{\partial L}{\partial w_\mathrm{h}} &= \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial w_\mathrm{h}} \\ &= \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial o_t} \frac{\partial g(h_t, w_\mathrm{o})}{\partial h_t} \frac{\partial h_t}{\partial w_\mathrm{h}}. \end{aligned} \tag{12.7.3}$$

(12.7.3) における積の最初と 2 番目の因子は簡単に計算できる。3 番目の因子 $\partial h_t / \partial w_\mathrm{h}$ が難所であり、パラメータ w_h が h_t に与える影響を再帰的に計算する必要がある。(12.7.1) の再帰的計算によれば、h_t は h_{t-1} と w_h の両方に依存し、h_{t-1} の計算もまた w_h に依存する。したがって、連鎖律を用いて h_t の w_h に関する全微分を評価すると、

$$\frac{\partial h_t}{\partial w_\mathrm{h}} = \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial w_\mathrm{h}} + \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_\mathrm{h}}. \tag{12.7.4}$$

上の勾配を導くために、3 つの系列 $\{a_t\}, \{b_t\}, \{c_t\}$ が $a_0 = 0$ かつ $t = 1, 2, \ldots$ に対して $a_t = b_t + c_t a_{t-1}$ を満たすと仮定する。すると $t \geq 1$ について、次が容易に示せる。

$$a_t = b_t + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} c_j \right) b_i. \tag{12.7.5}$$

a_t、b_t、c_t をそれぞれ

$$\begin{aligned} a_t &= \frac{\partial h_t}{\partial w_\mathrm{h}}, \\ b_t &= \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial w_\mathrm{h}}, \\ c_t &= \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial h_{t-1}}, \end{aligned} \tag{12.7.6}$$

に従って置き換えると、(12.7.4) の勾配計算は $a_t = b_t + c_t a_{t-1}$ を満たす。したがって、(12.7.5) により、(12.7.4) における再帰計算を

$$\frac{\partial h_t}{\partial w_\mathrm{h}} = \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial w_\mathrm{h}} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \frac{\partial f(x_j, h_{j-1}, w_\mathrm{h})}{\partial h_{j-1}} \right) \frac{\partial f(x_i, h_{i-1}, w_\mathrm{h})}{\partial w_\mathrm{h}}. \tag{12.7.7}$$

で取り除くことができる。

連鎖律を使って $\partial h_t / \partial w_\mathrm{h}$ を再帰的に計算できるが、t が大きいとこの連鎖は非常に長くなる。この問題に対処するいくつかの戦略を議論しよう。

全計算

1 つの考え方は、(12.7.7) の完全な和を計算することかもしれない。しかし、これは非常に遅く、勾配が爆発する可能性がある。というのも、初期条件のわずかな変化が結果に大きく影響しうるからである。つまり、初期条件の小さな変化が結果に不釣り合いな変化をもたらす、いわゆるバタフライ効果に似た現象が起こりえる。これは一般に望ましくない。結局のところ、私たちが求めているのは、よく一般化する頑健な推定量である。したがって、この戦略が実際に使われることはほとんどない。

タイムステップの切り詰め

別の方法として、(12.7.7) の和を τ ステップ後で打ち切ることができる。これが、ここまで議論してきた方法である。これは、和を $\partial h_{t-\tau} / \partial w_\mathrm{h}$ で単純に終了させることで、真の勾配の 近似を与える。実際には、これはかなりうまく機能する。これは一般に、切り詰めた時間を通した逆伝播と呼ばれる (Jaeger, 2002)。この結果の 1 つとして、モデルは長期的な結果よりも短期的な影響に主として注目するようになる。これは実際には 望ましいことであり、より単純で安定したモデルへと推定を偏らせるからである。

ランダム化された切り詰め

最後に、$\partial h_t / \partial w_\mathrm{h}$ を、期待値では正しいが系列を切り詰める確率変数で置き換えることができる。これは、あらかじめ定めた $0 \leq \pi_t \leq 1$ を持つ ξ_t の系列を用いて実現され、$P(\xi_t = 0) = 1 - \pi_t$ かつ $P(\xi_t = \pi_t^{-1}) = \pi_t$ となるため、$E[\xi_t] = 1$ である。これを用いて、(12.7.4) における勾配 $\partial h_t / \partial w_\mathrm{h}$ を

$$z_t = \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial w_\mathrm{h}} + \xi_t \frac{\partial f(x_t, h_{t-1}, w_\mathrm{h})}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_\mathrm{h}}. \tag{12.7.8}$$

で置き換える。

ξ_t の定義から、$E[z_t] = \partial h_t / \partial w_\mathrm{h}$ が成り立つ。$\xi_t = 0$ のときはいつでも、再帰計算はそのタイムステップ t で終了する。これにより、長さの異なる系列の重み付き和が得られ、長い系列はまれだが適切に重み付けされる。この考え方は Tallec and Ollivier (2017) によって提案された。

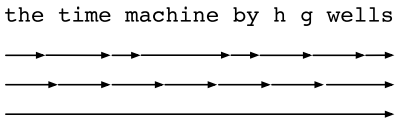


Fig. 12.7.1 RNN における勾配計算の戦略の比較。上から順に、ランダム化された切り詰め、通常の切り詰め、全計算。

Fig. 12.7.1 は、RNN に対して時間を通した逆伝播を用いて *The Time Machine* の最初の数文字を解析するときの 3 つの戦略を示している。

- 1 行目は、長さの異なる区間にテキストを分割するランダム化された切り詰めである。
- 2 行目は、同じ長さの部分系列にテキストを分割する通常の切り詰めである。これは、RNN の実験でこれまで行ってきた方法である。
- 3 行目は、計算上実行不可能な式につながる全時間逆伝播である。

残念ながら、理論的には魅力的であるものの、ランダム化された切り詰めは通常の切り詰めよりも大きく優れているわけではない。おそらく、いくつかの要因によるものである。第 1 に、過去へいくつかの逆伝播ステップを経た後の観測の影響は、実際には依存関係を捉えるのに十分である。第 2 に、分散の増加は、より多くのステップで勾配がより正確になるという事実を打ち消す。第 3 に、私たちは実際には、相互作用の範囲が短いモデルを 望んでいる。したがって、通常の切り詰めを用いた時間を通した逆伝播には、望ましい正則化効果がわずかにある。

12.7.2 時間を通した逆伝播の詳細

一般原理を議論したので、次に時間を通した逆伝播を詳しく見ていこう。Section 12.7.1 の解析とは対照的に、以下では、分解されたすべてのモデルパラメータに関する目的関数の勾配をどのように計算するかを示す。簡単のため、バイアスパラメータを持たず、隠れ層の活性化関数として恒等写像（$\phi(x) = x$）を用いる RNN を考える。タイムステップ t において、単一の例の入力と目標をそれぞれ $\mathbf{x}_t \in \mathbb{R}^d$ と y_t とする。隠れ状態 $\mathbf{h}_t \in \mathbb{R}^h$ と出力 $\mathbf{o}_t \in \mathbb{R}^q$ は次のように計算される。

$$\begin{aligned}\mathbf{h}_t &= \mathbf{W}_{\text{hx}}\mathbf{x}_t + \mathbf{W}_{\text{hh}}\mathbf{h}_{t-1},\\ \mathbf{o}_t &= \mathbf{W}_{\text{qh}}\mathbf{h}_t,\end{aligned} \tag{12.7.9}$$

ここで $\mathbf{W}_{\text{hx}} \in \mathbb{R}^{h\times d}$、$\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h\times h}$、$\mathbf{W}_{\text{qh}} \in \mathbb{R}^{q\times h}$ は重みパラメータである。時刻 t における損失を $l(\mathbf{o}_t, y_t)$ と表す。したがって、私たちの目的関数、すなわち系列の先頭から T タイムステップにわたる損失は

$$L = \frac{1}{T}\sum_{t=1}^{T} l(\mathbf{o}_t, y_t). \tag{12.7.10}$$

RNN の計算中にモデル変数とパラメータの間の依存関係を可視化するために、Fig. 12.7.2 に示すようなモデルの計算グラフを描くことができる。たとえば、3 番目のタイムステップの隠れ状態 $\mathbf{h}_3$ の計算は、モデルパラメータ $\mathbf{W}_{\text{hx}}$ と $\mathbf{W}_{\text{hh}}$、前のタイムステップの隠れ状態 $\mathbf{h}_2$、および現在のタイムステップの入力 $\mathbf{x}_3$ に依存する。

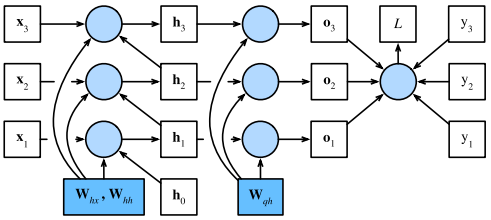


Fig. 12.7.2 3 タイムステップの RNN モデルにおける依存関係を示す計算グラフ。箱は変数（塗りつぶしなし）またはパラメータ（塗りつぶしあり）を表し、円は演算子を表す。

先ほど述べたように、Fig. 12.7.2 のモデルパラメータは $\mathbf{W}_{\text{hx}}$、$\mathbf{W}_{\text{hh}}$、$\mathbf{W}_{\text{qh}}$ である。一般に、このモデルの学習には、これらのパラメータに関する勾配計算 $\partial L/\partial \mathbf{W}_{\text{hx}}$、$\partial L/\partial \mathbf{W}_{\text{hh}}$、$\partial L/\partial \mathbf{W}_{\text{qh}}$ が必要である。Fig. 12.7.2 の依存関係に従って、矢印と逆向きにたどることで、順に勾配を計算して保存できる。連鎖律において、異なる形状を持つ行列、ベクトル、スカラーの積を柔軟に表現するために、Section 8.3 で説明したように prod 演算子を引き続き用いる。

まず、任意のタイムステップ t におけるモデル出力に関して目的関数を微分するのはかなり単純である。

$$\frac{\partial L}{\partial \mathbf{o}_t} = \frac{\partial l(\mathbf{o}_t, y_t)}{T \cdot \partial \mathbf{o}_t} \in \mathbb{R}^q. \tag{12.7.11}$$

これで、出力層のパラメータ $\mathbf{W}_{\text{qh}}$ に関する目的関数の勾配 $\partial L/\partial \mathbf{W}_{\text{qh}} \in \mathbb{R}^{q \times h}$ を計算できる。Fig. 12.7.2 に基づけば、目的関数 L は $\mathbf{o}_1, \ldots, \mathbf{o}_T$ を通じて $\mathbf{W}_{\text{qh}}$ に依存する。連鎖律を用いると、

$$\frac{\partial L}{\partial \mathbf{W}_{\text{qh}}} = \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{\text{qh}}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{o}_t} \mathbf{h}_t^\top, \tag{12.7.12}$$

ここで $\partial L/\partial \mathbf{o}_t$ は (12.7.11) で与えられる。

次に、Fig. 12.7.2 に示すように、最終タイムステップ T では、目的関数 L は出力 $\mathbf{o}_T$ を通じてのみ隠れ状態 $\mathbf{h}_T$ に依存する。したがって、連鎖律を用いて隠れ状態に関する勾配 $\partial L/\partial \mathbf{h}_T \in \mathbb{R}^h$ を容易に求められる。

$$\frac{\partial L}{\partial \mathbf{h}_T} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_T}, \frac{\partial \mathbf{o}_T}{\partial \mathbf{h}_T}\right) = \mathbf{W}_{\text{qh}}^\top \frac{\partial L}{\partial \mathbf{o}_T}. \tag{12.7.13}$$

任意のタイムステップ $t < T$ では事情が少し複雑になる。このとき、目的関数 L は $\mathbf{h}_{t+1}$ と $\mathbf{o}_t$ を通じて $\mathbf{h}_t$ に依存する。連鎖律に従うと、任意のタイムステップ $t < T$ における隠れ状態の勾配 $\partial L/\partial \mathbf{h}_t \in \mathbb{R}^h$ は再帰的に次のように計算でき

る。

$$\frac{\partial L}{\partial \mathbf{h}_t} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_{t+1}}, \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}\right) + \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\right) = \mathbf{W}_{\text{hh}}^\top \frac{\partial L}{\partial \mathbf{h}_{t+1}} + \mathbf{W}_{\text{qh}}^\top \frac{\partial L}{\partial \mathbf{o}_t}. \quad (12.7.14)$$

解析のために、任意のタイムステップ $1 \le t \le T$ について再帰計算を展開すると、

$$\frac{\partial L}{\partial \mathbf{h}_t} = \sum_{i=t}^{T} \left(\mathbf{W}_{\text{hh}}^\top\right)^{T-i} \mathbf{W}_{\text{qh}}^\top \frac{\partial L}{\partial \mathbf{o}_{T+t-i}}. \quad (12.7.15)$$

(12.7.15) からわかるように、この単純な線形例でさえ、長い系列モデルの重要な問題のいくつかをすでに示している。そこには、$\mathbf{W}_{\text{hh}}^\top$ の非常に大きな冪が含まれうるのである。その固有値が 1 より小さいと消失し、1 より大きいと発散する。これは数値的に不安定であり、消失勾配や爆発勾配として現れる。これに対処する 1 つの方法は、Section 12.7.1 で議論したように、タイムステップを計算上扱いやすい大きさで切り詰めることである。実際には、この切り詰めは、一定数のタイムステップの後で勾配を切り離すことでも実現できる。後ほど、長短期記憶（LSTM）のようなより洗練された系列モデルが、これをさらに緩和できることを見る。

最後に、Fig. 12.7.2 は、目的関数 L が、隠れ状態 $\mathbf{h}_1, \ldots, \mathbf{h}_T$ を通じて、隠れ層のモデルパラメータ $\mathbf{W}_{\text{hx}}$ と $\mathbf{W}_{\text{hh}}$ に依存することを示している。このようなパラメータに関する勾配 $\partial L/\partial \mathbf{W}_{\text{hx}} \in \mathbb{R}^{h \times d}$ と $\partial L/\partial \mathbf{W}_{\text{hh}} \in \mathbb{R}^{h \times h}$ を計算するには、連鎖律を適用して

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}_{\text{hx}}} &= \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{\text{hx}}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{x}_t^\top, \\ \frac{\partial L}{\partial \mathbf{W}_{\text{hh}}} &= \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{\text{hh}}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{h}_{t-1}^\top,\end{aligned} \quad (12.7.16)$$

ここで、(12.7.13) と (12.7.14) によって再帰的に計算される $\partial L/\partial \mathbf{h}_t$ が、数値安定性に影響する重要な量である。

時間を通した逆伝播は、RNN における逆伝播の適用そのものであるため、Section 8.3 で説明したように、RNN の学習では順伝播と時間を通した逆伝播を交互に行う。さらに、時間を通した逆伝播では、上記の勾配を順に計算して保存する。具体的には、保存された中間値を再利用して重複計算を避ける。たとえば、$\partial L/\partial \mathbf{h}_t$ を保存しておき、$\partial L/\partial \mathbf{W}_{\text{hx}}$ と $\partial L/\partial \mathbf{W}_{\text{hh}}$ の両方の計算に使う。

12.7.3 まとめ

時間を通した逆伝播は、隠れ状態を持つ系列モデルに逆伝播を適用したものにすぎない。通常の切り詰めやランダム化された切り詰めのような切り詰めは、計算上の都合と数値安定性のために必要である。行列の高次冪は、発散または消失する固有値を引き起こしえる。これは、爆発勾配や消失勾配として現れる。効率的に計算するために、時間を通した逆伝播の間、中間値はキャッシュされる。

12.7.4 演習

1. 対称行列 $\mathbf{M} \in \mathbb{R}^{n \times n}$ があり、その固有値を λ_i、対応する固有ベクトルを $\mathbf{v}_i$（$i = 1, \ldots, n$）とする。一般性を失わずに、$|\lambda_i| \geq |\lambda_{i+1}|$ の順に並んでいると仮定する。
   1. $\mathbf{M}^k$ の固有値が λ_i^k であることを示せ。
   2. ランダムなベクトル $\mathbf{x} \in \mathbb{R}^n$ に対して、高い確率で $\mathbf{M}^k\mathbf{x}$ が $\mathbf{M}$ の固有ベクトル $\mathbf{v}_1$ に非常によく整列することを証明せよ。この主張を形式化せよ。
   3. 上の結果は RNN の勾配にとって何を意味するか。
2. 勾配クリッピング以外に、再帰型ニューラルネットワークにおける勾配爆発に対処する他の方法を思いつくだろうか？

13 現代的な再帰型ニューラルネットワーク

前章では、再帰型ニューラルネットワーク（RNN）の背後にある主要な考え方を紹介した。
しかし、畳み込みニューラルネットワークと同様に、RNN のアーキテクチャにも非常に多くの革新があり、実際に成功を収めてきた複数の複雑な設計へと結実している。
特に、最も広く使われている設計では、消失勾配や爆発勾配に代表される、RNN が直面する悪名高い数値的不安定性を緩和するための仕組みが備わっている。
Section 12 では、勾配クリッピングという大まかなヒューリスティックを適用することで爆発勾配に対処した。
この場当たり的な手法は有効ではあるものの、消失勾配の問題は依然として残る。

この章では、2 本の論文に由来する、系列に対して最も成功している RNN アーキテクチャの背後にある主要な考え方を紹介する。
1 本目の *Long Short-Term Memory* (Hochreiter and Schmidhuber, 1997) は、メモリセルを導入する。これは、ネットワークの隠れ層にある従来のノードを置き換える計算単位である。
このメモリセルにより、ネットワークは以前の再帰型ネットワークが直面していた学習上の困難を克服できるようになる。
直感的には、メモリセルは、各メモリセルの内部状態の値を、重み 1 の再帰辺に沿って多数の連続した時間ステップにわたって伝播させることで、消失勾配問題を回避する。
一連の乗算ゲートは、メモリ状態にどの入力を通すかだけでなく、メモリ状態の内容をいつモデルの出力に影響させるべきかをネットワークが判断するのを助ける。

2 本目の *Bidirectional Recurrent Neural Networks* (Schuster and Paliwal, 1997) は、未来（後続の時間ステップ）と過去（前の時間ステップ）の両方からの情報を用いて、系列の任意の位置における出力を決定するアーキテクチャを導入する。
これは、過去の入力だけが出力に影響を与えられる従来のネットワークとは対照的である。

双方向 RNN は、自然言語処理における系列ラベリングタスクをはじめ、数多くのタスクで定番となっている。
幸いなことに、この 2 つの革新は相互排他的ではなく、音素分類 (Graves and Schmidhuber, 2005) や手書き認識 (Graves *et al.*, 2008) でうまく組み合わせられている。

この章の最初の節では、LSTM アーキテクチャ、ゲート付き再帰ユニット（GRU）と呼ばれるより軽量な版、双方向 RNN の背後にある主要な考え方、そして RNN 層を積み重ねて深い RNN を構成する方法について簡単に説明する。
その後、系列から系列へのタスクにおける RNN の応用を探り、エンコーダ–デコーダアーキテクチャや ビームサーチといった重要な考え方とともに機械翻訳を紹介する。

13.1 長短期記憶（LSTM）

最初の Elman 型 RNN が逆伝播を用いて学習された直後 (Elman, 1990)、長期依存関係の学習における問題（勾配消失と勾配爆発に起因する）が顕在化し、Bengio と Hochreiter がこの問題について議論した (Bengio *et al.*, 1994, Hochreiter *et al.*, 2001)。Hochreiter はこの問題を 1991 年の修士論文の時点で既に明確に述べていたが、その論文はドイツ語で書かれていたため、結果は広く知られていなかった。勾配クリッピングは勾配爆発には有効だが、勾配消失への対処には、より精巧な解決策が必要であるように見える。勾配消失に対処する最初期かつ最も成功した手法の一つが、Hochreiter and Schmidhuber (1997) による長短期記憶（LSTM）モデルである。LSTM は標準的な再帰ニューラルネットワークに似ているが、ここでは通常の再帰ノードがそれぞれ 記憶セルに置き換えられている。各記憶セルは 内部状態を含み、すなわち、重みが 1 に固定された自己接続の再帰辺を持つノードであり、勾配が消失も爆発もせずに多数の時間ステップをまたいで伝播できることを保証する。

「長短期記憶」という用語は、次の直感に由来する。単純な再帰ニューラルネットワークは、重みという形で 長期記憶を持つ。重みは学習中にゆっくり変化し、データに関する一般的な知識を符号化する。また、各ノードから次のノードへと受け渡される一時的な活性という形で 短期記憶も持つ。LSTM モデルは、記憶セルを通じて中間的な種類の記憶を導入する。記憶セルは、より単純なノードから特定の接続パターンで構成された複合ユニットであり、乗算ノードが新たに組み込まれている。

```
%load_ext d2lbook.tab
tab.interact_select(['mxnet', 'pytorch', 'tensorflow', 'jax'])
```

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import rnn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

13.1.1 ゲート付き記憶セル

各記憶セルは 内部状態と、いくつかの乗算ゲートを備えている。これらのゲートは、(i) 与えられた入力を内部状態に反映させるべきか（入力ゲート）、(ii) 内部状態を 0 に消去すべきか（忘却ゲート）、(iii) 与えられたニューロンの内部状態をセルの出力に反映させるべきか（出力ゲート）を決定する。

ゲート付き隠れ状態

素朴な RNN と LSTM の重要な違いは、後者が隠れ状態のゲーティングをサポートする点にある。これは、隠れ状態をいつ 更新すべきか、またいつ リセットすべきかのための専用機構があることを意味する。これらの機構は学習され、上で述べた懸念に対処する。たとえば、最初のトークンが非常に重要であれば、最初の観測以降は隠れ状態を更新しないように学習される。同様に、無関係な一時的観測を飛ばすように学習される。最後に、必要に応じて潜在状態をリセットするように学習される。これについては以下で詳しく説明する。

入力ゲート、忘却ゲート、出力ゲート

LSTM のゲートに入力されるデータは、現在の時間ステップの入力と前の時間ステップの隠れ状態であり、Fig. 13.1.1 に示されている。シグモイド活性化関数を持つ 3 つの全結合層が、入力ゲート、忘却ゲート、出力ゲートの値を計算する。シグモイド活性化の結果、3 つのゲートの値はすべて $(0, 1)$ の範囲に入る。さらに、入力ノードが必要であり、通常は *tanh* 活性化関数で計算される。直感的には、入力ゲートは入力ノードの値のどれだけを現在の記憶セル内部状態に加えるべきかを決定する。忘却ゲートは、現在の記憶を保持するか消去するかを決定する。そして 出力ゲートは、記憶セルが現在の時間ステップにおける出力へ影響を与えるべきかを決定する。

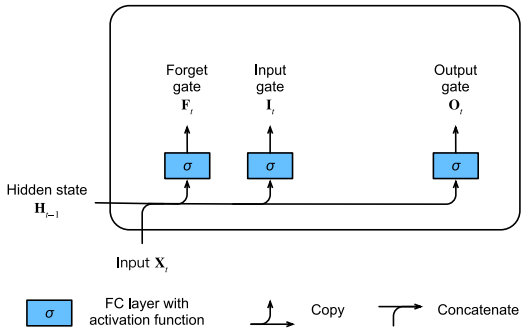


Fig. 13.1.1 LSTM モデルにおける入力ゲート、忘却ゲート、出力ゲートの計算。

数学的には、隠れユニット数が h、バッチサイズが n、入力数が d であるとする。したがって、入力は $\mathbf{X}_t \in \mathbb{R}^{n\times d}$、前の時間ステップの隠れ状態は $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$ である。対応して、時間ステップ t におけるゲートは次のように定義される。入力ゲートは $\mathbf{I}_t \in \mathbb{R}^{n\times h}$、忘却ゲートは $\mathbf{F}_t \in \mathbb{R}^{n\times h}$、出力ゲートは $\mathbf{O}_t \in \mathbb{R}^{n\times h}$ である。これらは次のように計算される。

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{\text{xi}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hi}} + \mathbf{b}_{\text{i}}),\\ \mathbf{F}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{\text{xf}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hf}} + \mathbf{b}_{\text{f}}),\\ \mathbf{O}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{\text{xo}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{ho}} + \mathbf{b}_{\text{o}}),\end{aligned} \tag{13.1.1}$$

ここで、$\mathbf{W}_{\text{xi}}, \mathbf{W}_{\text{xf}}, \mathbf{W}_{\text{xo}} \in \mathbb{R}^{d\times h}$ および $\mathbf{W}_{\text{hi}}, \mathbf{W}_{\text{hf}}, \mathbf{W}_{\text{ho}} \in \mathbb{R}^{h\times h}$ は重みパラメータであり、$\mathbf{b}_{\text{i}}, \mathbf{b}_{\text{f}}, \mathbf{b}_{\text{o}} \in \mathbb{R}^{1\times h}$ はバイアスパラメータである。加算の際にブロードキャスト（Section 5.1.4 を参照）が発生することに注意されたい。入力値を区間 $(0, 1)$ に写像するために、（Section 8.1 で導入した）シグモイド関数を用いる。

入力ノード

次に記憶セルを設計する。まだ各ゲートの働きを定義していないので、まず 入力ノード $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$ を導入する。その計算は上で述べた 3 つのゲートと似ているが、活性化関数として値域が $(-1, 1)$ の tanh 関数を用いる。これにより、時間ステップ t における次の式が得られる。

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{\text{xc}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hc}} + \mathbf{b}_{\text{c}}), \tag{13.1.2}$$

ここで、$\mathbf{W}_{\text{xc}} \in \mathbb{R}^{d\times h}$ および $\mathbf{W}_{\text{hc}} \in \mathbb{R}^{h\times h}$ は重みパラメータであり、$\mathbf{b}_{\text{c}} \in \mathbb{R}^{1\times h}$ はバイアスパラメータである。

入力ノードの簡単な図を Fig. 13.1.2 に示す。

記憶セル内部状態

LSTM では、入力ゲート $\mathbf{I}_t$ が $\tilde{\mathbf{C}}_t$ を通じて新しいデータをどれだけ考慮するかを制御し、忘却ゲート $\mathbf{F}_t$ が古いセル内部状態 $\mathbf{C}_{t-1} \in \mathbb{R}^{n\times h}$ のどれだけを保持する

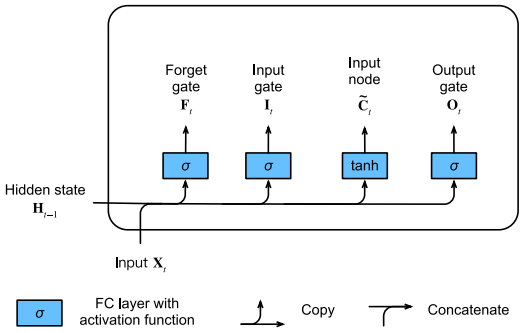


Fig. 13.1.2 LSTM モデルにおける入力ノードの計算。

かを扱う。Hadamard（要素ごと）積演算子 $\odot$ を用いると、次の更新式が得られる。

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t. \tag{13.1.3}$$

忘却ゲートが常に 1 で入力ゲートが常に 0 であれば、記憶セル内部状態 $\mathbf{C}_{t-1}$ は永遠に一定のまま保たれ、そのまま各後続時間ステップへ受け渡される。しかし、入力ゲートと忘却ゲートにより、モデルはこの値をいつ不変に保ち、いつ後続入力に応じて変化させるかを学習できる柔軟性を持つ。実際、この設計は勾配消失問題を緩和し、特に長い系列長を持つデータセットに対して、はるかに学習しやすいモデルを実現する。

これにより、Fig. 13.1.3 のフローダイアグラムが得られる。

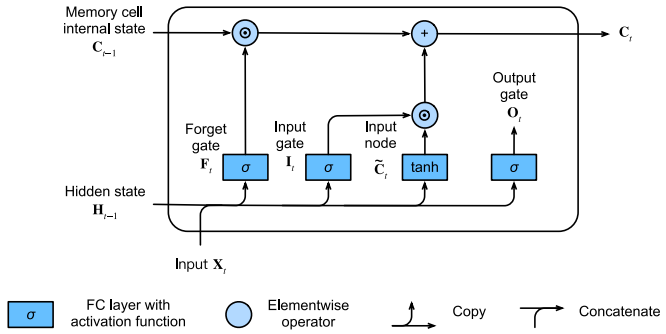


Fig. 13.1.3 LSTM モデルにおける記憶セル内部状態の計算。

隠れ状態

最後に、他の層から見える記憶セルの出力、すなわち隠れ状態 $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ をどのように計算するかを定義する必要がある。ここで出力ゲートが関与する。LSTM では、まず記憶セル内部状態に tanh を適用し、その後、今度は出力ゲートとの要素ごとの乗算を行う。これにより、$\mathbf{H}_t$ の値は常に区間 $(-1, 1)$ に収ま

る。

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t). \tag{13.1.4}$$

出力ゲートが 1 に近いときは、記憶セル内部状態が制約なく後続層に影響を与えることを許し、一方で出力ゲートの値が 0 に近いときは、現在の記憶がその時間ステップにおいてネットワークの他の層へ影響するのを防ぐ。記憶セルは、出力ゲートが 0 に近い値を取る限り、ネットワークの残りに影響を与えずに多数の時間ステップにわたって情報を蓄積でき、その後、出力ゲートが 0 に近い値から 1 に近い値へ切り替わると同時に、次の時間ステップで突然ネットワークに影響を与えることができる。データの流れの図示を Fig. 13.1.4 に示す。

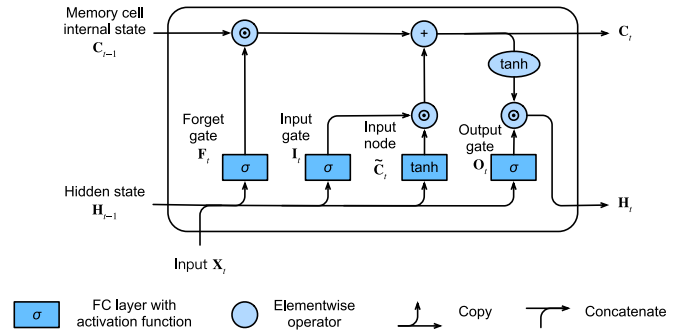


Fig. 13.1.4 LSTM モデルにおける隠れ状態の計算。

13.1.2 ゼロからの実装

それでは、LSTM をゼロから実装しよう。Section 12.5 の実験と同様に、まず *Time Machine* データセットを読み込む。

モデルパラメータの初期化

次に、モデルパラメータを定義し初期化する必要がある。前と同様に、ハイパーパラメータ `num_hiddens` が隠れユニット数を決定する。重みは標準偏差 0.01 のガウス分布に従って初期化し、バイアスは 0 に設定する。

```
class LSTMScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()

        if tab.selected('mxnet'):
            init_weight = lambda *shape: d2l.randn(*shape) * sigma
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              d2l.zeros(num_hiddens))
        if tab.selected('pytorch'):
```

(continues on next page)

(continued from previous page)

```
            init_weight = lambda *shape: nn.Parameter(d2l.randn(*shape) *␣
↪sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              nn.Parameter(d2l.zeros(num_hiddens)))
        if tab.selected('tensorflow'):
            init_weight = lambda *shape: tf.Variable(d2l.normal(shape) * sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              tf.Variable(d2l.zeros(num_hiddens)))

        self.W_xi, self.W_hi, self.b_i = triple()  # Input gate
        self.W_xf, self.W_hf, self.b_f = triple()  # Forget gate
        self.W_xo, self.W_ho, self.b_o = triple()  # Output gate
        self.W_xc, self.W_hc, self.b_c = triple()  # Input node
```

```
class LSTMScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()

        if tab.selected('mxnet'):
            init_weight = lambda *shape: d2l.randn(*shape) * sigma
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              d2l.zeros(num_hiddens))
        if tab.selected('pytorch'):
            init_weight = lambda *shape: nn.Parameter(d2l.randn(*shape) *␣
↪sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              nn.Parameter(d2l.zeros(num_hiddens)))
        if tab.selected('tensorflow'):
            init_weight = lambda *shape: tf.Variable(d2l.normal(shape) * sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              tf.Variable(d2l.zeros(num_hiddens)))

        self.W_xi, self.W_hi, self.b_i = triple()  # Input gate
        self.W_xf, self.W_hf, self.b_f = triple()  # Forget gate
        self.W_xo, self.W_ho, self.b_o = triple()  # Output gate
        self.W_xc, self.W_hc, self.b_c = triple()  # Input node
```

```
class LSTMScratch(d2l.Module):
    num_inputs: int
    num_hiddens: int
    sigma: float = 0.01

    def setup(self):
        init_weight = lambda name, shape: self.param(name,
                                                     nn.initializers.
↪normal(self.sigma),
                                                     shape)
```

(continues on next page)

(continued from previous page)

```
        triple = lambda name : (
            init_weight(f'W_x{name}', (self.num_inputs, self.num_hiddens)),
            init_weight(f'W_h{name}', (self.num_hiddens, self.num_hiddens)),
            self.param(f'b_{name}', nn.initializers.zeros, (self.num_hiddens)))

        self.W_xi, self.W_hi, self.b_i = triple('i')  # Input gate
        self.W_xf, self.W_hf, self.b_f = triple('f')  # Forget gate
        self.W_xo, self.W_ho, self.b_o = triple('o')  # Output gate
        self.W_xc, self.W_hc, self.b_c = triple('c')  # Input node
```

```
class LSTMScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()

        if tab.selected('mxnet'):
            init_weight = lambda *shape: d2l.randn(*shape) * sigma
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              d2l.zeros(num_hiddens))
        if tab.selected('pytorch'):
            init_weight = lambda *shape: nn.Parameter(d2l.randn(*shape) *␣
↪sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              nn.Parameter(d2l.zeros(num_hiddens)))
        if tab.selected('tensorflow'):
            init_weight = lambda *shape: tf.Variable(d2l.normal(shape) * sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              tf.Variable(d2l.zeros(num_hiddens)))

        self.W_xi, self.W_hi, self.b_i = triple()  # Input gate
        self.W_xf, self.W_hf, self.b_f = triple()  # Forget gate
        self.W_xo, self.W_ho, self.b_o = triple()  # Output gate
        self.W_xc, self.W_hc, self.b_c = triple()  # Input node
```

実際のモデルは上で述べたとおり、3 つのゲートと 1 つの入力ノードから構成される。なお、出力層に渡されるのは隠れ状態のみである。

```
@d2l.add_to_class(LSTMScratch)
def forward(self, inputs, H_C=None):
    if H_C is None:
        # Initial state with shape: (batch_size, num_hiddens)
        if tab.selected('mxnet'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          ctx=inputs.ctx)
            C = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          ctx=inputs.ctx)
        if tab.selected('pytorch'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
```

(continues on next page)

(continued from previous page)

```
            C = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
        if tab.selected('tensorflow'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens))
            C = d2l.zeros((inputs.shape[1], self.num_hiddens))
    else:
        H, C = H_C
    outputs = []
    for X in inputs:
        I = d2l.sigmoid(d2l.matmul(X, self.W_xi) +
                        d2l.matmul(H, self.W_hi) + self.b_i)
        F = d2l.sigmoid(d2l.matmul(X, self.W_xf) +
                        d2l.matmul(H, self.W_hf) + self.b_f)
        O = d2l.sigmoid(d2l.matmul(X, self.W_xo) +
                        d2l.matmul(H, self.W_ho) + self.b_o)
        C_tilde = d2l.tanh(d2l.matmul(X, self.W_xc) +
                           d2l.matmul(H, self.W_hc) + self.b_c)
        C = F * C + I * C_tilde
        H = O * d2l.tanh(C)
        outputs.append(H)
    return outputs, (H, C)
```

```
@d2l.add_to_class(LSTMScratch)
def forward(self, inputs, H_C=None):
    if H_C is None:
        # Initial state with shape: (batch_size, num_hiddens)
        if tab.selected('mxnet'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          ctx=inputs.ctx)
            C = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          ctx=inputs.ctx)
        if tab.selected('pytorch'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
            C = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
        if tab.selected('tensorflow'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens))
            C = d2l.zeros((inputs.shape[1], self.num_hiddens))
    else:
        H, C = H_C
    outputs = []
    for X in inputs:
        I = d2l.sigmoid(d2l.matmul(X, self.W_xi) +
                        d2l.matmul(H, self.W_hi) + self.b_i)
        F = d2l.sigmoid(d2l.matmul(X, self.W_xf) +
                        d2l.matmul(H, self.W_hf) + self.b_f)
        O = d2l.sigmoid(d2l.matmul(X, self.W_xo) +
                        d2l.matmul(H, self.W_ho) + self.b_o)
        C_tilde = d2l.tanh(d2l.matmul(X, self.W_xc) +
                           d2l.matmul(H, self.W_hc) + self.b_c)
        C = F * C + I * C_tilde
        H = O * d2l.tanh(C)
```

(continues on next page)

(continued from previous page)

```
        outputs.append(H)
    return outputs, (H, C)
```

```
@d2l.add_to_class(LSTMScratch)
def forward(self, inputs, H_C=None):
    # Use lax.scan primitive instead of looping over the
    # inputs, since scan saves time in jit compilation.
    def scan_fn(carry, X):
        H, C = carry
        I = d2l.sigmoid(d2l.matmul(X, self.W_xi) + (
            d2l.matmul(H, self.W_hi)) + self.b_i)
        F = d2l.sigmoid(d2l.matmul(X, self.W_xf) +
                        d2l.matmul(H, self.W_hf) + self.b_f)
        O = d2l.sigmoid(d2l.matmul(X, self.W_xo) +
                        d2l.matmul(H, self.W_ho) + self.b_o)
        C_tilde = d2l.tanh(d2l.matmul(X, self.W_xc) +
                           d2l.matmul(H, self.W_hc) + self.b_c)
        C = F * C + I * C_tilde
        H = O * d2l.tanh(C)
        return (H, C), H  # return carry, y

    if H_C is None:
        batch_size = inputs.shape[1]
        carry = jnp.zeros((batch_size, self.num_hiddens)), \
                jnp.zeros((batch_size, self.num_hiddens))
    else:
        carry = H_C

    # scan takes the scan_fn, initial carry state, xs with leading axis to be␣
↪scanned
    carry, outputs = jax.lax.scan(scan_fn, carry, inputs)
    return outputs, carry
```

```
@d2l.add_to_class(LSTMScratch)
def forward(self, inputs, H_C=None):
    if H_C is None:
        # Initial state with shape: (batch_size, num_hiddens)
        if tab.selected('mxnet'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          ctx=inputs.ctx)
            C = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          ctx=inputs.ctx)
        if tab.selected('pytorch'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
            C = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
        if tab.selected('tensorflow'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens))
            C = d2l.zeros((inputs.shape[1], self.num_hiddens))
    else:
        H, C = H_C
    outputs = []
```

(continues on next page)

(continued from previous page)

```
    for X in inputs:
        I = d2l.sigmoid(d2l.matmul(X, self.W_xi) +
                        d2l.matmul(H, self.W_hi) + self.b_i)
        F = d2l.sigmoid(d2l.matmul(X, self.W_xf) +
                        d2l.matmul(H, self.W_hf) + self.b_f)
        O = d2l.sigmoid(d2l.matmul(X, self.W_xo) +
                        d2l.matmul(H, self.W_ho) + self.b_o)
        C_tilde = d2l.tanh(d2l.matmul(X, self.W_xc) +
                           d2l.matmul(H, self.W_hc) + self.b_c)
        C = F * C + I * C_tilde
        H = O * d2l.tanh(C)
        outputs.append(H)
    return outputs, (H, C)
```

学習と予測

Section 12.5 の RNNLMScratch クラスをインスタンス化して、LSTM モデルを学習しよう。

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
if tab.selected('mxnet', 'pytorch', 'jax'):
    lstm = LSTMScratch(num_inputs=len(data.vocab), num_hiddens=32)
    model = d2l.RNNLMScratch(lstm, vocab_size=len(data.vocab), lr=4)
    trainer = d2l.Trainer(max_epochs=50, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        lstm = LSTMScratch(num_inputs=len(data.vocab), num_hiddens=32)
        model = d2l.RNNLMScratch(lstm, vocab_size=len(data.vocab), lr=4)
    trainer = d2l.Trainer(max_epochs=50, gradient_clip_val=1)
trainer.fit(model, data)
```

13.1.3 簡潔な実装

高水準 API を用いれば、LSTM モデルを直接インスタンス化できる。これにより、上で明示的に行ったすべての設定詳細がカプセル化される。ここまでで述べた多くの詳細について Python ではなくコンパイル済み演算子を使うため、コードは大幅に高速である。

```
class LSTM(d2l.RNN):
    def __init__(self, num_inputs, num_hiddens):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        self.rnn = nn.LSTM(num_inputs, num_hiddens)

    def forward(self, inputs, H_C=None):
        return self.rnn(inputs, H_C)
```

```
class LSTM(d2l.RNN):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, num_hiddens):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        self.rnn = rnn.LSTM(num_hiddens)

    def forward(self, inputs, H_C=None):
        if H_C is None: H_C = self.rnn.begin_state(
            inputs.shape[1], ctx=inputs.ctx)
        return self.rnn(inputs, H_C)
```

```
class LSTM(d2l.RNN):
    num_hiddens: int

    @nn.compact
    def __call__(self, inputs, H_C=None, training=False):
        if H_C is None:
            batch_size = inputs.shape[1]
            H_C = nn.OptimizedLSTMCell.initialize_carry(jax.random.PRNGKey(0),
                                                        (batch_size,),
                                                        self.num_hiddens)

        LSTM = nn.scan(nn.OptimizedLSTMCell, variable_broadcast="params",
                       in_axes=0, out_axes=0, split_rngs={"params": False})

        H_C, outputs = LSTM()(H_C, inputs)
        return outputs, H_C
```

```
class LSTM(d2l.RNN):
    def __init__(self, num_hiddens):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        self.rnn = tf.keras.layers.LSTM(
                num_hiddens, return_sequences=True,
                return_state=True, time_major=True)

    def forward(self, inputs, H_C=None):
        outputs, *H_C = self.rnn(inputs, H_C)
        return outputs, H_C
```

```
if tab.selected('pytorch'):
    lstm = LSTM(num_inputs=len(data.vocab), num_hiddens=32)
if tab.selected('mxnet', 'tensorflow', 'jax'):
    lstm = LSTM(num_hiddens=32)
if tab.selected('mxnet', 'pytorch', 'jax'):
    model = d2l.RNNLM(lstm, vocab_size=len(data.vocab), lr=4)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        model = d2l.RNNLM(lstm, vocab_size=len(data.vocab), lr=4)
trainer.fit(model, data)
```

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
'it has a four the preal is'
```

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
'it has all the time the ti'
```

```
model.predict('it has', 20, data.vocab, trainer.state.params)
```

```
'it has a mour the time tra'
```

```
model.predict('it has', 20, data.vocab)
```

```
'it has i the thing the thi'
```

LSTM は、非自明な状態制御を備えた潜在変数自己回帰モデルの典型例である。長年にわたり、その多様な変種が提案されてきた。たとえば、多層化、残差接続、さまざまな正則化手法などである。しかし、系列の長距離依存性のため、LSTM や他の系列モデル（GRU など）の学習はかなりコストが高い。後の章では、場合によっては利用できる Transformer などの代替モデルに出会うことになる。

13.1.4 まとめ

LSTM は 1997 年に発表されたが、2000 年代半ばの予測コンペティションでいくつかの勝利を収めたことで大きく注目され、2011 年から 2017 年に Transformer モデルが台頭するまで、系列学習の支配的なモデルとなった。Transformer でさえ、その重要なアイデアのいくつかは、LSTM によって導入されたアーキテクチャ設計上の革新に負っている。

LSTM には 3 種類のゲートがある。入力ゲート、忘却ゲート、出力ゲートであり、情報の流れを制御する。LSTM の隠れ層出力には隠れ状態と記憶セル内部状態が含まれる。出力層に渡されるのは隠れ状態のみであり、記憶セル内部状態は完全に内部にとどまる。LSTM は勾配消失と勾配爆発を緩和できる。

13.1.5 演習

1. ハイパーパラメータを調整し、実行時間、パープレキシティ、出力系列への影響を分析せよ。

2. 文字列の系列ではなく、適切な単語を生成するには、モデルをどのように変更する必要があるか。
3. 与えられた隠れ次元に対して、GRU、LSTM、通常の RNN の計算コストを比較せよ。特に学習時と推論時のコストに注意せよ。
4. 候補記憶セルは tanh 関数を用いて値域を −1 から 1 に保つのに、なぜ隠れ状態でも再び tanh 関数を用いて出力値域を −1 から 1 に保つ必要があるのか。
5. 文字系列予測ではなく、時系列予測のための LSTM モデルを実装せよ。

13.2 ゲート付き再帰ユニット（GRU）

2010 年代に RNN、特に LSTM アーキテクチャ（Section 13.1）が急速に普及するにつれて、多くの研究者が、内部状態と乗算的なゲーティング機構を組み込むという重要なアイデアを保ちながら、計算を高速化することを目指して、より単純化したアーキテクチャの実験を始めた。ゲート付き再帰ユニット（GRU）(Cho *et al.*, 2014) は、LSTM のメモリセルを簡潔にした版を提供し、しばしば同等の性能を達成しつつ、計算がより高速であるという利点を持つ (Chung *et al.*, 2014)。

```
%load_ext d2lbook.tab
tab.interact_select(['mxnet', 'pytorch', 'tensorflow', 'jax'])
```

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import rnn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

13.2.1 リセットゲートと更新ゲート

ここでは、LSTM の 3 つのゲートを 2 つに置き換える。それがリセットゲートと更新ゲートである。LSTM と同様に、これらのゲートにはシグモイド活性化が与えられ、その値は区間 $(0, 1)$ に制限される。直感的には、リセットゲートは、前の状態のうちどれだけをまだ記憶しておきたいかを制御する。同様に、更新ゲートは、新しい状態のうちどれだけが古い状態の単なるコピーであるかを制御できるようにする。Fig. 13.2.1 は、現在の時刻の入力と前時刻の隠れ状態が与えられたときの、GRU におけるリセットゲートと更新ゲートの両方への入力を示している。ゲートの出力は、シグモイド活性化関数を持つ 2 つの全結合層によって与えられる。

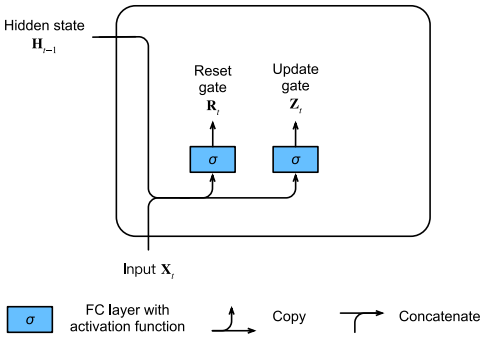


Fig. 13.2.1 GRU モデルにおけるリセットゲートと更新ゲートの計算。

数学的には、ある時刻 t に対して、入力がミニバッチ $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ （例の数 $= n$、入力数 $= d$）であり、前時刻の隠れ状態が $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ （隠れユニット数 $= h$）であるとする。このとき、リセットゲート $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ と更新ゲート $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$ は次のように計算される：

$$\begin{aligned} \mathbf{R}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{\text{xr}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hr}} + \mathbf{b}_{\text{r}}), \\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{\text{xz}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hz}} + \mathbf{b}_{\text{z}}), \end{aligned} \tag{13.2.1}$$

ここで、$\mathbf{W}_{\text{xr}}, \mathbf{W}_{\text{xz}} \in \mathbb{R}^{d \times h}$ および $\mathbf{W}_{\text{hr}}, \mathbf{W}_{\text{hz}} \in \mathbb{R}^{h \times h}$ は重みパラメータであり、$\mathbf{b}_{\text{r}}, \mathbf{b}_{\text{z}} \in \mathbb{R}^{1 \times h}$ はバイアスパラメータである。

13.2.2 候補隠れ状態

次に、リセットゲート $\mathbf{R}_t$ を (12.4.5) における通常の更新機構と統合し、時刻 t における次の 候補隠れ状態 $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ を得る：

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{\text{xh}} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}), \tag{13.2.2}$$

ここで、$\mathbf{W}_{\text{xh}} \in \mathbb{R}^{d \times h}$ および $\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h \times h}$ は重みパラメータ、$\mathbf{b}_{\text{h}} \in \mathbb{R}^{1 \times h}$ はバイアスであり、記号 $\odot$ はハダマード積（要素ごとの積）演算子である。ここでは tanh 活性化関数を用いる。

これはまだ候補である。なぜなら、更新ゲートの働きをまだ組み込む必要があるからである。(12.4.5) と比較すると、前の状態の影響は、(13.2.2) における $\mathbf{R}_t$ と $\mathbf{H}_{t-1}$ の要素ごとの乗算によって抑えられるようになっている。リセットゲ

ート $\mathbf{R}_t$ の各要素が 1 に近いとき、(12.4.5) のような通常の RNN が復元される。リセットゲート $\mathbf{R}_t$ の各要素が 0 に近いとき、候補隠れ状態は $\mathbf{X}_t$ を入力とする MLP の結果になる。したがって、既存の隠れ状態はデフォルト値にリセットされる。

Fig. 13.2.2 は、リセットゲートを適用した後の計算の流れを示している。

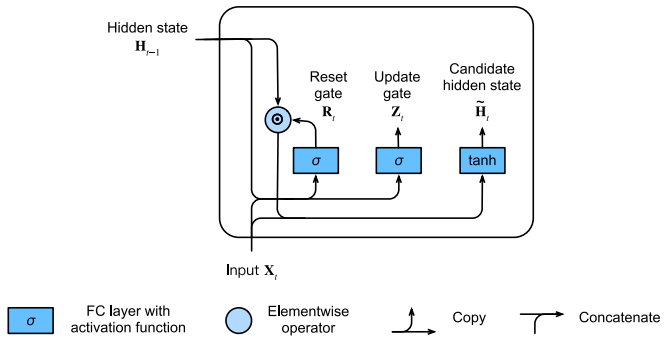


Fig. 13.2.2 GRU モデルにおける候補隠れ状態の計算。

13.2.3 隠れ状態

最後に、更新ゲート $\mathbf{Z}_t$ の効果を組み込む必要がある。これは、新しい隠れ状態 $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ が古い状態 $\mathbf{H}_{t-1}$ にどの程度一致するかと、新しい候補状態 $\tilde{\mathbf{H}}_t$ にどの程度似るかを決定する。更新ゲート $\mathbf{Z}_t$ は、この目的のために、$\mathbf{H}_{t-1}$ と $\tilde{\mathbf{H}}_t$ の要素ごとの凸結合を取るだけで用いることができる。これにより、GRU の最終的な更新式は次のようになる：

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t. \tag{13.2.3}$$

更新ゲート $\mathbf{Z}_t$ が 1 に近いときは、単に古い状態を保持する。この場合、$\mathbf{X}_t$ からの情報は無視され、依存関係の連鎖において時刻 t が実質的にスキップされる。対照的に、$\mathbf{Z}_t$ が 0 に近いときは、新しい潜在状態 $\mathbf{H}_t$ は候補潜在状態 $\tilde{\mathbf{H}}_t$ に近づく。Fig. 13.2.3 は、更新ゲートが働いた後の計算の流れを示している。

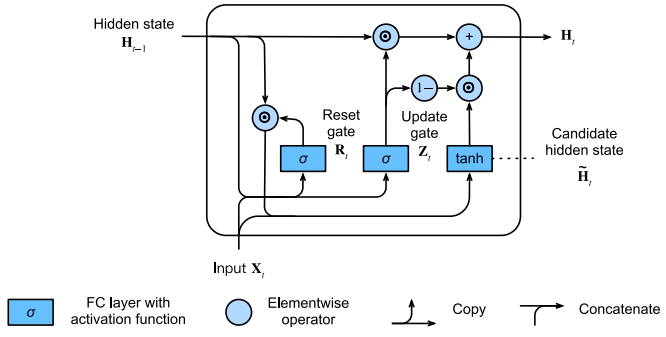


Fig. 13.2.3 GRU モデルにおける隠れ状態の計算。

要約すると、GRU には次の 2 つの特徴がある：

- リセットゲートは、系列における短期依存関係の捕捉に役立つ。
- 更新ゲートは、系列における長期依存関係の捕捉に役立つ。

13.2.4 ゼロからの実装

GRU モデルをよりよく理解するために、ゼロから実装してみよう。

モデルパラメータの初期化

最初のステップは、モデルパラメータを初期化することである。重みは標準偏差が `sigma` のガウス分布からサンプリングし、バイアスは 0 に設定する。ハイパーパラメータ `num_hiddens` は隠れユニット数を定義する。更新ゲート、リセットゲート、候補隠れ状態に関するすべての重みとバイアスをインスタンス化する。

```
class GRUScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()

        if tab.selected('mxnet'):
            init_weight = lambda *shape: d2l.randn(*shape) * sigma
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              d2l.zeros(num_hiddens))
        if tab.selected('pytorch'):
            init_weight = lambda *shape: nn.Parameter(d2l.randn(*shape) *␣
↪sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              nn.Parameter(d2l.zeros(num_hiddens)))
        if tab.selected('tensorflow'):
            init_weight = lambda *shape: tf.Variable(d2l.normal(shape) * sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              tf.Variable(d2l.zeros(num_hiddens)))

        self.W_xz, self.W_hz, self.b_z = triple()  # Update gate
        self.W_xr, self.W_hr, self.b_r = triple()  # Reset gate
        self.W_xh, self.W_hh, self.b_h = triple()  # Candidate hidden state
```

```
class GRUScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()

        if tab.selected('mxnet'):
            init_weight = lambda *shape: d2l.randn(*shape) * sigma
            triple = lambda: (init_weight(num_inputs, num_hiddens),
```

(continues on next page)

(continued from previous page)

```
                            init_weight(num_hiddens, num_hiddens),
                            d2l.zeros(num_hiddens))
        if tab.selected('pytorch'):
            init_weight = lambda *shape: nn.Parameter(d2l.randn(*shape) *␣
↪sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                            init_weight(num_hiddens, num_hiddens),
                            nn.Parameter(d2l.zeros(num_hiddens)))
        if tab.selected('tensorflow'):
            init_weight = lambda *shape: tf.Variable(d2l.normal(shape) * sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                            init_weight(num_hiddens, num_hiddens),
                            tf.Variable(d2l.zeros(num_hiddens)))

        self.W_xz, self.W_hz, self.b_z = triple()  # Update gate
        self.W_xr, self.W_hr, self.b_r = triple()  # Reset gate
        self.W_xh, self.W_hh, self.b_h = triple()  # Candidate hidden state
```

```
class GRUScratch(d2l.Module):
    num_inputs: int
    num_hiddens: int
    sigma: float = 0.01

    def setup(self):
        init_weight = lambda name, shape: self.param(name,
                                                     nn.initializers.
↪normal(self.sigma),
                                                     shape)
        triple = lambda name : (
            init_weight(f'W_x{name}', (self.num_inputs, self.num_hiddens)),
            init_weight(f'W_h{name}', (self.num_hiddens, self.num_hiddens)),
            self.param(f'b_{name}', nn.initializers.zeros, (self.num_hiddens)))

        self.W_xz, self.W_hz, self.b_z = triple('z')  # Update gate
        self.W_xr, self.W_hr, self.b_r = triple('r')  # Reset gate
        self.W_xh, self.W_hh, self.b_h = triple('h')  # Candidate hidden state
```

```
class GRUScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()

        if tab.selected('mxnet'):
            init_weight = lambda *shape: d2l.randn(*shape) * sigma
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                            init_weight(num_hiddens, num_hiddens),
                            d2l.zeros(num_hiddens))
        if tab.selected('pytorch'):
            init_weight = lambda *shape: nn.Parameter(d2l.randn(*shape) *␣
↪sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                            init_weight(num_hiddens, num_hiddens),
                            nn.Parameter(d2l.zeros(num_hiddens)))
```

(continues on next page)

(continued from previous page)

```
        if tab.selected('tensorflow'):
            init_weight = lambda *shape: tf.Variable(d2l.normal(shape) * sigma)
            triple = lambda: (init_weight(num_inputs, num_hiddens),
                              init_weight(num_hiddens, num_hiddens),
                              tf.Variable(d2l.zeros(num_hiddens)))

        self.W_xz, self.W_hz, self.b_z = triple()  # Update gate
        self.W_xr, self.W_hr, self.b_r = triple()  # Reset gate
        self.W_xh, self.W_hh, self.b_h = triple()  # Candidate hidden state
```

モデルの定義

これで GRU の順伝播計算を定義する準備が整った。その構造は基本的な RNN セルと同じだが、更新式がより複雑である。

```
@d2l.add_to_class(GRUScratch)
def forward(self, inputs, H=None):
    if H is None:
        # Initial state with shape: (batch_size, num_hiddens)
        if tab.selected('mxnet'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          ctx=inputs.ctx)
        if tab.selected('pytorch'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
        if tab.selected('tensorflow'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens))
    outputs = []
    for X in inputs:
        Z = d2l.sigmoid(d2l.matmul(X, self.W_xz) +
                        d2l.matmul(H, self.W_hz) + self.b_z)
        R = d2l.sigmoid(d2l.matmul(X, self.W_xr) +
                        d2l.matmul(H, self.W_hr) + self.b_r)
        H_tilde = d2l.tanh(d2l.matmul(X, self.W_xh) +
                           d2l.matmul(R * H, self.W_hh) + self.b_h)
        H = Z * H + (1 - Z) * H_tilde
        outputs.append(H)
    return outputs, H
```

```
@d2l.add_to_class(GRUScratch)
def forward(self, inputs, H=None):
    if H is None:
        # Initial state with shape: (batch_size, num_hiddens)
        if tab.selected('mxnet'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          ctx=inputs.ctx)
        if tab.selected('pytorch'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
        if tab.selected('tensorflow'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens))
```

(continues on next page)

(continued from previous page)

```
    outputs = []
    for X in inputs:
        Z = d2l.sigmoid(d2l.matmul(X, self.W_xz) +
                        d2l.matmul(H, self.W_hz) + self.b_z)
        R = d2l.sigmoid(d2l.matmul(X, self.W_xr) +
                        d2l.matmul(H, self.W_hr) + self.b_r)
        H_tilde = d2l.tanh(d2l.matmul(X, self.W_xh) +
                           d2l.matmul(R * H, self.W_hh) + self.b_h)
        H = Z * H + (1 - Z) * H_tilde
        outputs.append(H)
    return outputs, H
```

```
@d2l.add_to_class(GRUScratch)
def forward(self, inputs, H=None):
    # Use lax.scan primitive instead of looping over the
    # inputs, since scan saves time in jit compilation
    def scan_fn(H, X):
        Z = d2l.sigmoid(d2l.matmul(X, self.W_xz) + d2l.matmul(H, self.W_hz) +
                        self.b_z)
        R = d2l.sigmoid(d2l.matmul(X, self.W_xr) +
                        d2l.matmul(H, self.W_hr) + self.b_r)
        H_tilde = d2l.tanh(d2l.matmul(X, self.W_xh) +
                           d2l.matmul(R * H, self.W_hh) + self.b_h)
        H = Z * H + (1 - Z) * H_tilde
        return H, H  # return carry, y

    if H is None:
        batch_size = inputs.shape[1]
        carry = jnp.zeros((batch_size, self.num_hiddens))
    else:
        carry = H

    # scan takes the scan_fn, initial carry state, xs with leading axis to be␣
↪scanned
    carry, outputs = jax.lax.scan(scan_fn, carry, inputs)
    return outputs, carry
```

```
@d2l.add_to_class(GRUScratch)
def forward(self, inputs, H=None):
    if H is None:
        # Initial state with shape: (batch_size, num_hiddens)
        if tab.selected('mxnet'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          ctx=inputs.ctx)
        if tab.selected('pytorch'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
        if tab.selected('tensorflow'):
            H = d2l.zeros((inputs.shape[1], self.num_hiddens))
    outputs = []
    for X in inputs:
        Z = d2l.sigmoid(d2l.matmul(X, self.W_xz) +
                        d2l.matmul(H, self.W_hz) + self.b_z)
```

(continues on next page)

(continued from previous page)

```
        R = d2l.sigmoid(d2l.matmul(X, self.W_xr) +
                        d2l.matmul(H, self.W_hr) + self.b_r)
        H_tilde = d2l.tanh(d2l.matmul(X, self.W_xh) +
                           d2l.matmul(R * H, self.W_hh) + self.b_h)
        H = Z * H + (1 - Z) * H_tilde
        outputs.append(H)
    return outputs, H
```

学習

タイムマシンデータセット上で言語モデルを学習する方法は、Section 12.5 とまったく同じである。

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
if tab.selected('mxnet', 'pytorch', 'jax'):
    gru = GRUScratch(num_inputs=len(data.vocab), num_hiddens=32)
    model = d2l.RNNLMScratch(gru, vocab_size=len(data.vocab), lr=4)
    trainer = d2l.Trainer(max_epochs=50, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        gru = GRUScratch(num_inputs=len(data.vocab), num_hiddens=32)
        model = d2l.RNNLMScratch(gru, vocab_size=len(data.vocab), lr=4)
    trainer = d2l.Trainer(max_epochs=50, gradient_clip_val=1)
trainer.fit(model, data)
```

13.2.5 簡潔な実装

高水準 API では、GRU モデルを直接インスタンス化できる。これにより、上で明示的に記述したすべての設定の詳細がカプセル化される。

```
class GRU(d2l.RNN):
    def __init__(self, num_inputs, num_hiddens):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.rnn = rnn.GRU(num_hiddens)
        if tab.selected('pytorch'):
            self.rnn = nn.GRU(num_inputs, num_hiddens)
        if tab.selected('tensorflow'):
            self.rnn = tf.keras.layers.GRU(num_hiddens, return_sequences=True,
                                           return_state=True)
```

```
class GRU(d2l.RNN):
    def __init__(self, num_inputs, num_hiddens):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.rnn = rnn.GRU(num_hiddens)
        if tab.selected('pytorch'):
```

(continues on next page)

(continued from previous page)

```
            self.rnn = nn.GRU(num_inputs, num_hiddens)
        if tab.selected('tensorflow'):
            self.rnn = tf.keras.layers.GRU(num_hiddens, return_sequences=True,
                                           return_state=True)
```

```
class GRU(d2l.RNN):
    num_hiddens: int

    @nn.compact
    def __call__(self, inputs, H=None, training=False):
        if H is None:
            batch_size = inputs.shape[1]
            H = nn.GRUCell.initialize_carry(jax.random.PRNGKey(0),
                                            (batch_size,), self.num_hiddens)

        GRU = nn.scan(nn.GRUCell, variable_broadcast="params",
                      in_axes=0, out_axes=0, split_rngs={"params": False})

        H, outputs = GRU()(H, inputs)
        return outputs, H
```

```
class GRU(d2l.RNN):
    def __init__(self, num_inputs, num_hiddens):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.rnn = rnn.GRU(num_hiddens)
        if tab.selected('pytorch'):
            self.rnn = nn.GRU(num_inputs, num_hiddens)
        if tab.selected('tensorflow'):
            self.rnn = tf.keras.layers.GRU(num_hiddens, return_sequences=True,
                                           return_state=True)
```

このコードは、Python ではなくコンパイル済みオペレータを使用するため、学習時に大幅に高速である。

```
if tab.selected('mxnet', 'pytorch', 'tensorflow'):
    gru = GRU(num_inputs=len(data.vocab), num_hiddens=32)
if tab.selected('jax'):
    gru = GRU(num_hiddens=32)
if tab.selected('mxnet', 'pytorch', 'jax'):
    model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=4)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=4)
trainer.fit(model, data)
```

学習後、訓練セット上のパープレキシティと、与えられた接頭辞に続く予測系列を出力する。

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
'it has is the time the pec'
```

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
'it has the the the the the'
```

```
model.predict('it has', 20, data.vocab, trainer.state.params)
```

```
'it has and the the the the'
```

```
model.predict('it has', 20, data.vocab)
```

```
'it has the the the the the'
```

13.2.6 まとめ

LSTM と比べると、GRU は同程度の性能を達成しつつ、計算量が軽い傾向がある。一般に、単純な RNN と比べると、ゲート付き RNN は、LSTM や GRU と同様に、時間ステップ間の距離が大きい系列における依存関係をよりよく捉えられる。GRU は、リセットゲートがオンになると、基本的な RNN を極限ケースとして含む。また、更新ゲートをオンにすることで部分系列をスキップすることもできる。

13.2.7 演習

1. 時刻 t' の入力だけを使って、時刻 $t > t'$ の出力を予測したいとする。各時刻におけるリセットゲートと更新ゲートの最適な値は何か。
2. ハイパーパラメータを調整し、実行時間、パープレキシティ、出力系列への影響を分析せよ。
3. `rnn.RNN` と `rnn.GRU` の実装について、実行時間、パープレキシティ、出力文字列を相互に比較せよ。
4. GRU の一部だけ、たとえばリセットゲートだけ、あるいは更新ゲートだけを実装するとどうなるか。

13.3 深層再帰ニューラルネットワーク

これまで、系列入力、単一の隠れ RNN 層、および出力層からなるネットワークの定義に焦点を当ててきた。
任意の時刻の入力とそれに対応する出力の間に隠れ層が 1 つしかないにもかかわらず、これらのネットワークには深さがあると言える。
最初の時刻の入力は、最終時刻 T の出力（しばしば 100 ステップや 1000 ステップも後）に影響を与えることができる。
これらの入力は、最終出力に到達するまでに再帰層を T 回通過する。
しかし、私たちはしばしば、ある時刻における入力と同じ時刻における出力の間の複雑な関係を表現する能力も保持したいと考える。
そのため、RNN は時間方向だけでなく、入力から出力への方向にも深いように構成されることがよくある。
これは、MLP や深層 CNN の開発で既に出会ってきた、まさに深さの概念である。

この種の深い RNN を構築する標準的な方法は驚くほど単純である。RNN を上に積み重ねればよいのである。
長さ T の系列が与えられると、最初の RNN は長さ T の出力系列を生成する。
これらは次の RNN 層への入力を構成する。
この短い節では、この設計パターンを示し、そのようなスタック型 RNN をどのように実装するかの簡単な例を示す。
以下の Fig. 13.3.1 では、L 個の隠れ層を持つ深い RNN を示す。
各隠れ状態は系列入力を受け取り、系列出力を生成する。
さらに、各時刻における任意の RNN セル（Fig. 13.3.1 の白い箱）は、同じ層の前時刻の値と、同じ時刻の前の層の値の両方に依存する。

形式的には、時刻 t においてミニバッチ入力 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$（例の数 $= n$、各例の入力数 $= d$）があるとする。
同じ時刻において、l^{th} 隠れ層（$l = 1, \ldots, L$）の隠れ状態を $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$（隠れユニット数 $= h$）とし、出力層変数を $\mathbf{O}_t \in \mathbb{R}^{n \times q}$（出力数 $= q$）とする。
$\mathbf{H}_t^{(0)} = \mathbf{X}_t$ とおくと、活性化関数 ϕ_l を用いる l^{th} 隠れ層の隠れ状態は次のように計算される。

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)}\mathbf{W}_{\text{xh}}^{(l)} + \mathbf{H}_{t-1}^{(l)}\mathbf{W}_{\text{hh}}^{(l)} + \mathbf{b}_{\text{h}}^{(l)}), \tag{13.3.1}$$

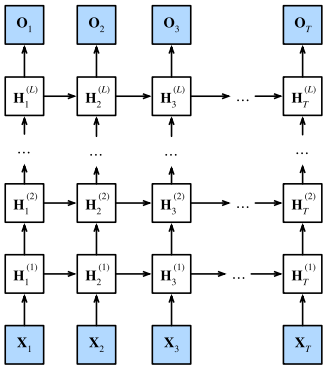


Fig. 13.3.1 深い RNN のアーキテクチャ。

ここで、重み $\mathbf{W}_{\text{xh}}^{(l)} \in \mathbb{R}^{h \times h}$ と $\mathbf{W}_{\text{hh}}^{(l)} \in \mathbb{R}^{h \times h}$、およびバイアス $\mathbf{b}_{\text{h}}^{(l)} \in \mathbb{R}^{1 \times h}$ は、l^{th} 隠れ層のモデルパラメータである。

最後に、出力層の計算は最終の L^{th} 隠れ層の隠れ状態のみに基づく。

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}, \tag{13.3.2}$$

ここで、重み $\mathbf{W}_{\text{hq}} \in \mathbb{R}^{h \times q}$ とバイアス $\mathbf{b}_{\text{q}} \in \mathbb{R}^{1 \times q}$ は出力層のモデルパラメータである。

MLP と同様に、隠れ層の数 L と隠れユニット数 h は調整可能なハイパーパラメータである。
一般的な RNN 層の幅（h）は $(64, 2056)$ の範囲にあり、一般的な深さ（L）は $(1, 8)$ の範囲にある。
さらに、(13.3.1) の隠れ状態計算を LSTM や GRU のものに置き換えることで、深いゲート付き RNN を容易に得ることができる。

```
%load_ext d2lbook.tab
tab.interact_select('mxnet', 'pytorch', 'tensorflow', 'jax')
```

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import rnn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

13.3.1 ゼロからの実装

ゼロから多層 RNN を実装するには、各層をそれぞれ独自の学習可能パラメータを持つ RNNScratch インスタンスとして扱えばよいである。

```
class StackedRNNScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, num_layers, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.rnns = nn.Sequential(*[d2l.RNNScratch(
            num_inputs if i==0 else num_hiddens, num_hiddens, sigma)
                                    for i in range(num_layers)])
```

```
class StackedRNNScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, num_layers, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.rnns = [d2l.RNNScratch(num_inputs if i==0 else num_hiddens,
                                    num_hiddens, sigma)
                     for i in range(num_layers)]
```

```
class StackedRNNScratch(d2l.Module):
    num_inputs: int
    num_hiddens: int
    num_layers: int
    sigma: float = 0.01

    def setup(self):
        self.rnns = [d2l.RNNScratch(self.num_inputs if i==0 else self.num_
↪hiddens,
                                    self.num_hiddens, self.sigma)
                     for i in range(self.num_layers)]
```

```
class StackedRNNScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, num_layers, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.rnns = [d2l.RNNScratch(num_inputs if i==0 else num_hiddens,
                                    num_hiddens, sigma)
                     for i in range(num_layers)]
```

多層の順伝播計算は、単純に層ごとに順伝播を行うだけである。

```
@d2l.add_to_class(StackedRNNScratch)
def forward(self, inputs, Hs=None):
    outputs = inputs
    if Hs is None: Hs = [None] * self.num_layers
    for i in range(self.num_layers):
        outputs, Hs[i] = self.rnns[i](outputs, Hs[i])
        outputs = d2l.stack(outputs, 0)
    return outputs, Hs
```

例として、*The Time Machine* データセット（Section 12.5 と同じ）上で深い GRU モデルを学習する。
簡単のため、層数は 2 に設定する。

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
if tab.selected('mxnet', 'pytorch', 'jax'):
    rnn_block = StackedRNNScratch(num_inputs=len(data.vocab),
                                  num_hiddens=32, num_layers=2)
    model = d2l.RNNLMScratch(rnn_block, vocab_size=len(data.vocab), lr=2)
    trainer = d2l.Trainer(max_epochs=100, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        rnn_block = StackedRNNScratch(num_inputs=len(data.vocab),
                                      num_hiddens=32, num_layers=2)
        model = d2l.RNNLMScratch(rnn_block, vocab_size=len(data.vocab), lr=2)
    trainer = d2l.Trainer(max_epochs=100, gradient_clip_val=1)
trainer.fit(model, data)
```

13.3.2 簡潔な実装

幸いなことに、RNN の複数層を実装するために必要な多くの実務的な詳細は、高レベル API で容易に利用できる。
ここでの簡潔な実装では、そのような組み込み機能を使用する。
このコードは、Section 13.2 で以前に使ったものを一般化したもので、デフォルトの 1 層ではなく、層数を明示的に指定できるようにしている。

```
class GRU(d2l.RNN):  #@save
    """The multilayer GRU model."""
    def __init__(self, num_inputs, num_hiddens, num_layers, dropout=0):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        self.rnn = nn.GRU(num_inputs, num_hiddens, num_layers,
                          dropout=dropout)
```

```
class GRU(d2l.RNN):  #@save
    """The multilayer GRU model."""
    def __init__(self, num_hiddens, num_layers, dropout=0):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=dropout)
```

```
class GRU(d2l.RNN):  #@save
    """The multilayer GRU model."""
    num_hiddens: int
    num_layers: int
    dropout: float = 0

    @nn.compact
    def __call__(self, X, state=None, training=False):
        outputs = X
        new_state = []
        if state is None:
            batch_size = X.shape[1]
            state = [nn.GRUCell.initialize_carry(jax.random.PRNGKey(0),
                    (batch_size,), self.num_hiddens)] * self.num_layers

        GRU = nn.scan(nn.GRUCell, variable_broadcast="params",
                      in_axes=0, out_axes=0, split_rngs={"params": False})

        # 最後の GRU 層を除く各 GRU 層の後に dropout 層を導入する
        for i in range(self.num_layers - 1):
            layer_i_state, X = GRU()(state[i], outputs)
            new_state.append(layer_i_state)
            X = nn.Dropout(self.dropout, deterministic=not training)(X)

        # dropout なしの最終 GRU 層
        out_state, X = GRU()(state[-1], X)
        new_state.append(out_state)
        return X, jnp.array(new_state)
```

```
class GRU(d2l.RNN):  #@save
    """The multilayer GRU model."""
    def __init__(self, num_hiddens, num_layers, dropout=0):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        gru_cells = [tf.keras.layers.GRUCell(num_hiddens, dropout=dropout)
                     for _ in range(num_layers)]
        self.rnn = tf.keras.layers.RNN(gru_cells, return_sequences=True,
                                       return_state=True, time_major=True)

    def forward(self, X, state=None):
        outputs, *state = self.rnn(X, state)
        return outputs, state
```

ハイパーパラメータの選択などのアーキテクチャ上の決定は、Section 13.2 のものと非常によく似ている。

異なるトークンの数、すなわち vocab_size と同じ数の入力と出力を選ぶ。
隠れユニット数は引き続き 32 である。
唯一の違いは、ここでは num_layers の値を指定することで、単純ではない数の隠れ層を選択することである。

```
if tab.selected('tensorflow', 'jax'):
    gru = GRU(num_hiddens=32, num_layers=2)
if tab.selected('pytorch'):
    gru = GRU(num_inputs=len(data.vocab), num_hiddens=32, num_layers=2)
if tab.selected('pytorch', 'jax'):
    model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=2)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=2)
trainer.fit(model, data)
```

```
gru = GRU(num_hiddens=32, num_layers=2)
model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=2)

# 実行には 1 時間以上かかる（MXNet 側の修正待ち）
# trainer.fit(model, data)
# model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
if tab.selected('tensorflow', 'jax'):
    gru = GRU(num_hiddens=32, num_layers=2)
if tab.selected('pytorch'):
    gru = GRU(num_inputs=len(data.vocab), num_hiddens=32, num_layers=2)
if tab.selected('pytorch', 'jax'):
    model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=2)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=2)
trainer.fit(model, data)
```

```
if tab.selected('tensorflow', 'jax'):
    gru = GRU(num_hiddens=32, num_layers=2)
if tab.selected('pytorch'):
    gru = GRU(num_inputs=len(data.vocab), num_hiddens=32, num_layers=2)
if tab.selected('pytorch', 'jax'):
    model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=2)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        model = d2l.RNNLM(gru, vocab_size=len(data.vocab), lr=2)
trainer.fit(model, data)
```

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

```
'it has experiment a mather'
```

```
model.predict('it has', 20, data.vocab, trainer.state.params)
```

```
'it has i some to the time '
```

```
model.predict('it has', 20, data.vocab)
```

```
'it has the time traveller '
```

13.3.3 まとめ

深い RNN では、隠れ状態情報は現在の層の次の時刻と、次の層の現在の時刻へと受け渡される。

LSTM、GRU、あるいは通常の RNN など、深い RNN にはさまざまな種類がある。

便利なことに、これらのモデルはすべて、深層学習フレームワークの高レベル API の一部として利用できる。

モデルの初期化には注意が必要である。

全体として、深い RNN は適切に収束させるために、学習率やクリッピングなど、かなりの作業を必要とする。

13.3.4 演習

1. GRU を LSTM に置き換え、精度と学習速度を比較せよ。
2. 学習データを増やして複数の本を含めるようにせよ。困惑度をどこまで下げられるか。
3. テキストをモデル化する際に、異なる著者のソースを組み合わせたいと思うか。なぜそれがよい考えなのか。何が問題になりうるか。

13.4 双方向リカレントニューラルネットワーク

これまで、系列学習タスクの作業例として言語モデルを扱ってきた。そこでは、系列内のそれ以前のすべてのトークンが与えられたときに、次のトークンを予測することを目指す。この状況では、左側の文脈のみに条件づければよく、そのため標準的な RNN の一方向の連鎖が適しているように見える。しか

し、他にも多くの系列学習タスクがあり、そこでは各時刻での予測を左側と右側の両方の文脈に条件づけることがまったく問題ない。たとえば、品詞判定を考えてみよう。ある単語に対応する品詞を評価するときに、なぜ両方向の文脈を考慮しないのだろうか。

もう 1 つよくあるタスクとして、実際に関心のあるタスクでモデルをファインチューニングする前の事前学習として有用なことが多いものに、テキスト文書中のランダムなトークンをマスクし、欠落したトークンの値を予測するよう系列モデルを学習させる、というものがある。空欄の後に何が続くかによって、欠落トークンの有力候補は大きく変わることに注意されたい。

- I am ___.
- I am ___ hungry.
- I am ___ hungry, and I can eat half a pig.

最初の文では、「happy」が有力な候補に見える。2 つ目の文では「not」と「very」のどちらもありえそうであるが、3 つ目の文では「not」は適切ではなさそうである。

幸いなことに、単純な手法で任意の一方向 RNN を双方向 RNN に変換できる (Schuster and Paliwal, 1997)。同じ入力に対して、逆方向に連結された 2 つの一方向 RNN 層を実装するだけです（Fig. 13.4.1）。最初の RNN 層では、最初の入力は $\mathbf{x}_1$ で最後の入力は $\mathbf{x}_T$ であるが、2 つ目の RNN 層では、最初の入力は $\mathbf{x}_T$ で最後の入力は $\mathbf{x}_1$ である。この双方向 RNN 層の出力を得るには、2 つの基礎となる一方向 RNN 層の対応する出力を単純に連結する。

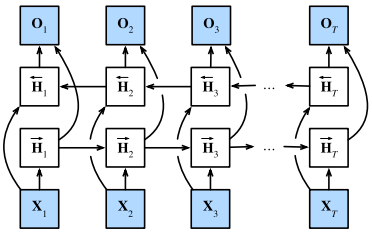


Fig. 13.4.1 双方向 RNN のアーキテクチャ。

形式的には、任意の時刻 t について、ミニバッチ入力 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ （例の数 = n、各例における入力数 = d）を考え、隠れ層の活性化関数を ϕ とする。双方向アーキテクチャでは、この時刻における順方向と逆方向の隠れ状態はそれぞれ $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ と $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ である。ここで h は隠れユニット数である。順方向と逆方向の隠れ状態の更新は次のようになる。

$$\begin{aligned}\overrightarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{\text{xh}}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{\text{hh}}^{(f)} + \mathbf{b}_{\text{h}}^{(f)}),\\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{\text{xh}}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{\text{hh}}^{(b)} + \mathbf{b}_{\text{h}}^{(b)}),\end{aligned} \tag{13.4.1}$$

ここで、重み $\mathbf{W}_{\text{xh}}^{(f)} \in \mathbb{R}^{d \times h}, \mathbf{W}_{\text{hh}}^{(f)} \in \mathbb{R}^{h \times h}, \mathbf{W}_{\text{xh}}^{(b)} \in \mathbb{R}^{d \times h}$, and $\mathbf{W}_{\text{hh}}^{(b)} \in \mathbb{R}^{h \times h}$、およびバイアス $\mathbf{b}_{\text{h}}^{(f)} \in \mathbb{R}^{1 \times h}$ と $\mathbf{b}_{\text{h}}^{(b)} \in \mathbb{R}^{1 \times h}$ はすべてモデルパラメータである。

次に、順方向と逆方向の隠れ状態 $\overrightarrow{\mathbf{H}}_t$ と $\overleftarrow{\mathbf{H}}_t$ を連結して、出力層に入力する隠れ状態 $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ を得る。複数の隠れ層を持つ深い双方向 RNN では、このような情報は次の双方向層への 入力として渡される。最後に、出力層は出力 $\mathbf{O}_t \in \mathbb{R}^{n \times q}$（出力数 $= q$）を計算する。

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}. \tag{13.4.2}$$

ここで、重み行列 $\mathbf{W}_{\text{hq}} \in \mathbb{R}^{2h \times q}$ とバイアス $\mathbf{b}_{\text{q}} \in \mathbb{R}^{1 \times q}$ は出力層のモデルパラメータである。厳密には、2 つの方向で異なる数の隠れユニットを持たせることもできるが、実際にはこの設計が選ばれることはほとんどない。それでは、双方向 RNN の簡単な実装を示す。

```
%load_ext d2lbook.tab
tab.interact_select('mxnet', 'pytorch', 'tensorflow', 'jax')
```

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import npx, np
from mxnet.gluon import rnn
npx.set_np()
```

```
from d2l import jax as d2l
from jax import numpy as jnp
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

13.4.1 ゼロからの実装

双方向 RNN をゼロから実装したい場合は、別々の学習可能パラメータを持つ 2 つの一方向 `RNNScratch` インスタンスを含めればよいである。

```
class BiRNNScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.f_rnn = d2l.RNNScratch(num_inputs, num_hiddens, sigma)
        self.b_rnn = d2l.RNNScratch(num_inputs, num_hiddens, sigma)
        self.num_hiddens *= 2  # The output dimension will be doubled
```

```
class BiRNNScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        self.save_hyperparameters()
        self.f_rnn = d2l.RNNScratch(num_inputs, num_hiddens, sigma)
        self.b_rnn = d2l.RNNScratch(num_inputs, num_hiddens, sigma)
        self.num_hiddens *= 2  # The output dimension will be doubled
```

```
class BiRNNScratch(d2l.Module):
    num_inputs: int
    num_hiddens: int
    sigma: float = 0.01

    def setup(self):
        self.f_rnn = d2l.RNNScratch(num_inputs, num_hiddens, sigma)
        self.b_rnn = d2l.RNNScratch(num_inputs, num_hiddens, sigma)
        self.num_hiddens *= 2  # The output dimension will be doubled
```

```
class BiRNNScratch(d2l.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.f_rnn = d2l.RNNScratch(num_inputs, num_hiddens, sigma)
        self.b_rnn = d2l.RNNScratch(num_inputs, num_hiddens, sigma)
        self.num_hiddens *= 2  # The output dimension will be doubled
```

順方向と逆方向の RNN の状態は別々に更新され、これら 2 つの RNN の出力は連結される。

```
@d2l.add_to_class(BiRNNScratch)
def forward(self, inputs, Hs=None):
    f_H, b_H = Hs if Hs is not None else (None, None)
    f_outputs, f_H = self.f_rnn(inputs, f_H)
    b_outputs, b_H = self.b_rnn(reversed(inputs), b_H)
    outputs = [d2l.concat((f, b), -1) for f, b in zip(
        f_outputs, reversed(b_outputs))]
    return outputs, (f_H, b_H)
```

13.4.2 簡潔な実装

高水準 API を使えば、双方向 RNN をより簡潔に実装できる。ここでは GRU モデルを例として取り上げる。

```
class BiGRU(d2l.RNN):
    def __init__(self, num_inputs, num_hiddens):
        d2l.Module.__init__(self)
        self.save_hyperparameters()
        if tab.selected('mxnet'):
            self.rnn = rnn.GRU(num_hiddens, bidirectional=True)
        if tab.selected('pytorch'):
            self.rnn = nn.GRU(num_inputs, num_hiddens, bidirectional=True)
        self.num_hiddens *= 2
```

13.4.3 まとめ

双方向 RNN では、各時刻の隠れ状態は、現在の時刻より前と後のデータの両方によって同時に決まりる。双方向 RNN は主に、系列の符号化や、双方向の文脈が与えられた観測の推定に有用である。双方向 RNN は、長い勾配連鎖のために学習コストが非常に高くなる。

13.4.4 演習

1. 異なる方向で異なる数の隠れユニットを使う場合、$\mathbf{H}_t$ の形状はどのように変わるか。
2. 複数の隠れ層を持つ双方向 RNN を設計せよ。
3. 自然言語では多義性が一般的である。たとえば、単語“bank”は“i went to the bank to deposit cash”と“i went to the bank to sit down”という文脈で異なる意味を持つ。文脈系列と単語が与えられたときに、その単語の正しい文脈におけるベクトル表現を返すようなニューラルネットワークモデルをどのように設計できるか？多義性の扱いにはどのような種類のニューラルアーキテクチャが好まれるか。

13.5 機械翻訳とデータセット

現代の RNN に対する広範な関心を引き起こした主要なブレークスルーの一つは、応用分野である統計的 機械翻訳における大きな進歩でした。ここでは、モデルにある言語の文を与え、対応する別の言語の文を予測させる。なお、ここでは文の長さが異なる場合があり、また、2 つの文における対応する単語が同じ順序で現れないこともある。これは、2 つの言語の文法構造の違いによるものである。

このような、2 つの「整列していない」系列の間の写像という性質をもつ問題は他にも多くある。例としては、対話のプロンプトから応答への写像や、質問から答えへの写像が挙げられる。広く言えば、このような問題は *sequence-to-sequence*（seq2seq）問題と呼ばれ、本章の残りと Section 14 の大部分で私たちが扱うテーマである。

この節では、機械翻訳問題と、後続の例で用いるデータセットの例を紹介する。何十年もの間、言語間翻訳の統計的定式化は研究者がニューラルネットワークによる手法を実用化する以前から人気があった (Brown *et al.*, 1990, Brown *et al.*, 1988)（これらの手法はしばしばまとめて *neural machine translation* という用語で呼ばれていた）。

まず、データを処理するための新しいコードが必要である。Section 12.3 で見た言語モデリングとは異なり、ここでは各例が 2 つの別々のテキスト系列からなり、一方はソース言語、もう一方は（翻訳された）ターゲット言語である。以

下のコード片では、前処理済みデータを学習用のミニバッチに読み込む方法を示す。

```
from d2l import torch as d2l
import torch
import os
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
import os
npx.set_np()
```

```
from d2l import jax as d2l
from jax import numpy as jnp
import os
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
import os
```

13.5.1 データセットのダウンロードと前処理

80 

まず、Tatoeba Project のバイリンガル文ペア[80] からなる英仏データセットをダウンロードする。データセットの各行はタブ区切りのペアで、英語のテキスト系列（*source*）と翻訳されたフランス語のテキスト系列（*target*）から構成される。なお、各テキスト系列は 1 文だけの場合もあれば、複数文からなる段落の場合もある。

```
class MTFraEng(d2l.DataModule):  #@save
    """The English-French dataset."""
    def _download(self):
        d2l.extract(d2l.download(
            d2l.DATA_URL+'fra-eng.zip', self.root,
            '94646ad1522d915e7b0f9296181140edcf86a4f5'))
        with open(self.root + '/fra-eng/fra.txt', encoding='utf-8') as f:
            return f.read()
```

```
data = MTFraEng()
raw_text = data._download()
print(raw_text[:75])
```

```
Downloading ../data/fra-eng.zip from http://d2l-data.s3-accelerate.amazonaws.
```

(continues on next page)

(continued from previous page)

```
↪com/fra-eng.zip...
Go. Va !
Hi. Salut !
Run!        Cours !
Run!        Courez !
Who?        Qui ?
Wow!        Ça alors !
```

データセットをダウンロードした後、生のテキストデータに対していくつかの前処理を行う。たとえば、改行なしスペースを通常のスペースに置き換え、大文字を小文字に変換し、単語と句読点の間にスペースを挿入する。

```
@d2l.add_to_class(MTFraEng)  #@save
def _preprocess(self, text):
    # Replace non-breaking space with space
    text = text.replace('\u202f', ' ').replace('\xa0', ' ')
    # Insert space between words and punctuation marks
    no_space = lambda char, prev_char: char in ',.!?' and prev_char != ' '
    out = [' ' + char if i > 0 and no_space(char, text[i - 1]) else char
           for i, char in enumerate(text.lower())]
    return ''.join(out)
```

```
text = data._preprocess(raw_text)
print(text[:80])
```

```
go .        va !
hi .        salut !
run !       cours !
run !       courez !
who ?       qui ?
wow !       ça alors !
```

13.5.2 トークン化

Section 12.3 における文字単位のトークン化とは異なり、機械翻訳ではここでは単語単位のトークン化を用いるのが一般的です（現在の最先端モデルでは、より複雑なトークン化手法が使われている）。以下の `_tokenize` メソッドは、最初の `max_examples` 個のテキスト系列ペアをトークン化する。各トークンは単語または句読点のいずれかである。各系列の末尾には、系列の終端を示す特別な“<eos>”トークンを追加する。モデルがトークンを 1 つずつ生成しながら系列を予測する場合、“<eos>”トークンの生成は出力系列が完了したことを示唆する。最終的に、以下のメソッドは `src` と `tgt` という 2 つのトークン列のリストを返す。具体的には、`src[i]` はソース言語（ここでは英語）の i^{th} テキスト系列のトークン列であり、`tgt[i]` はターゲット言語（ここではフランス語）のトークン列である。

```
@d2l.add_to_class(MTFraEng)  #@save
def _tokenize(self, text, max_examples=None):
    src, tgt = [], []
    for i, line in enumerate(text.split('\n')):
        if max_examples and i > max_examples: break
        parts = line.split('\t')
        if len(parts) == 2:
            # Skip empty tokens
            src.append([t for t in f'{parts[0]} <eos>'.split(' ') if t])
            tgt.append([t for t in f'{parts[1]} <eos>'.split(' ') if t])
    return src, tgt
```

```
src, tgt = data._tokenize(text)
src[:6], tgt[:6]
```

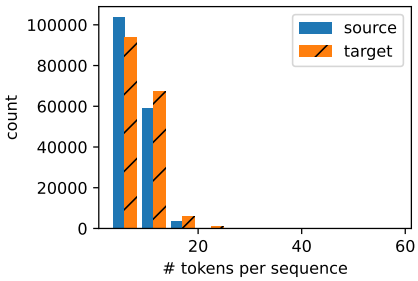
```
([['go', '.', '<eos>'],
  ['hi', '.', '<eos>'],
  ['run', '!', '<eos>'],
  ['run', '!', '<eos>'],
  ['who', '?', '<eos>'],
  ['wow', '!', '<eos>']],
 [['va', '!', '<eos>'],
  ['salut', '!', '<eos>'],
  ['cours', '!', '<eos>'],
  ['courez', '!', '<eos>'],
  ['qui', '?', '<eos>'],
  ['ça', 'alors', '!', '<eos>']])
```

テキスト系列ごとのトークン数のヒストグラムを描いてみよう。この単純な英仏データセットでは、ほとんどのテキスト系列は 20 トークン未満である。

```
#@save
def show_list_len_pair_hist(legend, xlabel, ylabel, xlist, ylist):
    """Plot the histogram for list length pairs."""
    d2l.set_figsize()
    _, _, patches = d2l.plt.hist(
        [[len(l) for l in xlist], [len(l) for l in ylist]])
    d2l.plt.xlabel(xlabel)
    d2l.plt.ylabel(ylabel)
    for patch in patches[1].patches:
        patch.set_hatch('/')
    d2l.plt.legend(legend)
```

```
show_list_len_pair_hist(['source', 'target'], '# tokens per sequence',
                        'count', src, tgt);
```

13.5.3 固定長系列の読み込み



言語モデリングでは、各例の系列が、1 文の一部であれ複数文にまたがる区間であれ、固定長であったことを思い出されたい。これは Section 12.3 の num_steps（時間ステップ数またはトークン数）引数で指定されていた。機械翻訳では、各例はソース系列とターゲット系列のペアであり、2 つのテキスト系列の長さは異なっていてもかまわない。

計算効率のため、切り詰めと パディングによって、テキスト系列のミニバッチを一度に処理することができる。同じミニバッチ内のすべての系列が同じ長さ num_steps を持つと仮定する。あるテキスト系列のトークン数が num_steps より少ない場合は、その末尾に特別な“<pad>”トークンを長さが num_steps に達するまで追加し続ける。そうでない場合は、最初の num_steps 個のトークンだけを取り出し、残りを切り捨てる。このようにして、すべてのテキスト系列は同じ形状のミニバッチに読み込めるよう、同じ長さになる。さらに、パディングトークンを除いたソース系列の長さも記録する。この情報は、後で扱ういくつかのモデルで必要になる。

機械翻訳データセットは言語のペアから構成されるため、ソース言語とターゲット言語の両方について別々に 2 つの語彙を構築できる。単語単位のトークン化では、語彙サイズは文字単位のトークン化よりもかなり大きくなる。これを緩和するために、ここでは 2 回未満しか現れない低頻度トークンをすべて同じ未知（“<unk>”）トークンとして扱う。後で説明するように（Fig. 13.7.1）、ターゲット系列で学習するとき、デコーダの出力（ラベルトークン）はデコーダ入力（ターゲットトークン）を 1 トークン分ずらしたものと同じにできる。また、特別な系列開始“<bos>”トークンは、ターゲット系列を予測する際の最初の入力トークンとして使われる（Fig. 13.7.3）。

```
@d2l.add_to_class(MTFraEng)  #@save
def __init__(self, batch_size, num_steps=9, num_train=512, num_val=128):
    super(MTFraEng, self).__init__()
    self.save_hyperparameters()
    self.arrays, self.src_vocab, self.tgt_vocab = self._build_arrays(
        self._download())
```

```
@d2l.add_to_class(MTFraEng)  #@save
def _build_arrays(self, raw_text, src_vocab=None, tgt_vocab=None):
    def _build_array(sentences, vocab, is_tgt=False):
```

(continues on next page)

(continued from previous page)

```
        pad_or_trim = lambda seq, t: (
            seq[:t] if len(seq) > t else seq + ['<pad>'] * (t - len(seq)))
        sentences = [pad_or_trim(s, self.num_steps) for s in sentences]
        if is_tgt:
            sentences = [['<bos>'] + s for s in sentences]
        if vocab is None:
            vocab = d2l.Vocab(sentences, min_freq=2)
        array = d2l.tensor([vocab[s] for s in sentences])
        valid_len = d2l.reduce_sum(
            d2l.astype(array != vocab['<pad>'], d2l.int32), 1)
        return array, vocab, valid_len
    src, tgt = self._tokenize(self._preprocess(raw_text),
                              self.num_train + self.num_val)
    src_array, src_vocab, src_valid_len = _build_array(src, src_vocab)
    tgt_array, tgt_vocab, _ = _build_array(tgt, tgt_vocab, True)
    return ((src_array, tgt_array[:,:-1], src_valid_len, tgt_array[:,1:]),
            src_vocab, tgt_vocab)
```

13.5.4 データセットの読み込み

最後に、データ反復子を返す get_dataloader メソッドを定義する。

```
@d2l.add_to_class(MTFraEng)  #@save
def get_dataloader(self, train):
    idx = slice(0, self.num_train) if train else slice(self.num_train, None)
    return self.get_tensorloader(self.arrays, train, idx)
```

英仏データセットから最初のミニバッチを読み込んでみよう。

```
data = MTFraEng(batch_size=3)
src, tgt, src_valid_len, label = next(iter(data.train_dataloader()))
print('source:', d2l.astype(src, d2l.int32))
print('decoder input:', d2l.astype(tgt, d2l.int32))
print('source len excluding pad:', d2l.astype(src_valid_len, d2l.int32))
print('label:', d2l.astype(label, d2l.int32))
```

```
source: tensor([[ 59,   2,   3,   4,   4,   4,   4,   4,   4],
        [ 22,  37,   2,   3,   4,   4,   4,   4,   4],
        [ 21, 111,   2,   3,   4,   4,   4,   4,   4]], dtype=torch.int32)
decoder input: tensor([[  3, 201,   0,   4,   5,   5,   5,   5,   5],
        [  3,   6,   2,   4,   5,   5,   5,   5,   5],
        [  3,   6,   0,   4,   5,   5,   5,   5,   5]], dtype=torch.int32)
source len excluding pad: tensor([3, 4, 4], dtype=torch.int32)
label: tensor([[201,   0,   4,   5,   5,   5,   5,   5,   5],
        [  6,   2,   4,   5,   5,   5,   5,   5,   5],
        [  6,   0,   4,   5,   5,   5,   5,   5,   5]], dtype=torch.int32)
```

上の _build_arrays メソッドで処理されたソース系列とターゲット系列のペアを（文字列形式で）示す。

```
@d2l.add_to_class(MTFraEng)  #@save
def build(self, src_sentences, tgt_sentences):
    raw_text = '\n'.join([src + '\t' + tgt for src, tgt in zip(
        src_sentences, tgt_sentences)])
    arrays, _, _ = self._build_arrays(
        raw_text, self.src_vocab, self.tgt_vocab)
    return arrays
```

```
src, tgt, _,  _ = data.build(['hi .'], ['salut .'])
print('source:', data.src_vocab.to_tokens(d2l.astype(src[0], d2l.int32)))
print('target:', data.tgt_vocab.to_tokens(d2l.astype(tgt[0], d2l.int32)))
```

```
source: ['hi', '.', '<eos>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '
↪<pad>']
target: ['<bos>', 'salut', '.', '<eos>', '<pad>', '<pad>', '<pad>', '<pad>', '
↪<pad>']
```

13.5.5 まとめ

自然言語処理において、機械翻訳とは、ソース言語のテキスト文字列を表す系列から、妥当な翻訳を表す ターゲット言語の文字列へ自動的に写像する課題を指す。単語単位のトークン化を用いると、語彙サイズは文字単位のトークン化よりもかなり大きくなるが、系列長ははるかに短くなる。大きな語彙サイズを緩和するために、低頻度トークンを何らかの「未知」トークンとして扱うことができる。テキスト系列は切り詰めとパディングによって同じ長さにそろえ、同じ形状のミニバッチとして読み込めるようにする。現代の実装では、パディングによる無駄な計算を避けるために、長さの近い系列をバケット化することがよくある。

13.5.6 演習

1. `_tokenize` メソッドの `max_examples` 引数にさまざまな値を試してみよ。これはソース言語とターゲット言語の語彙サイズにどのような影響を与えるか。
2. 中国語や日本語のように、単語境界を示す記号（たとえばスペース）がない言語もある。そのような場合でも単語単位のトークン化は良い考えであろうか。なぜそう言えるか、あるいはなぜそう言えないかを説明せよ。

13.6 エンコーダ–デコーダアーキテクチャ

一般に、機械翻訳 (Section 13.5) のようなシーケンス・ツー・シーケンス問題では、入力と出力は長さがさまざまで、互いに対応が取れていない。この種のデータを扱う標準的な方法は、2 つの主要な構成要素からなる エンコーダ–デコー

ダアーキテクチャ (Fig. 13.6.1) を設計することである。すなわち、可変長の系列を入力として受け取る エンコーダと、エンコードされた入力とターゲット系列の左側の文脈を受け取り、ターゲット系列における次のトークンを予測する条件付き言語モデルとして働く デコーダである。

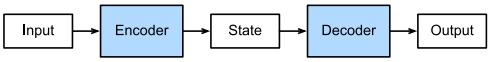


Fig. 13.6.1 エンコーダ–デコーダアーキテクチャ。

英仏機械翻訳を例に考えてみよう。英語の入力系列“They”, “are”, “watching”, “.”が与えられると、このエンコーダ–デコーダアーキテクチャはまず可変長の入力を状態にエンコードし、その後、その状態をデコードして翻訳された系列をトークンごとに出力として生成する。“Ils”, “regardent”, “.”。エンコーダ–デコーダアーキテクチャは後続の節でさまざまなシーケンス・ツー・シーケンスモデルの基盤となるため、この節ではこのアーキテクチャを後で実装するためのインターフェースに落とし込みる。

```
from d2l import torch as d2l
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet.gluon import nn
```

```
from d2l import jax as d2l
from flax import linen as nn
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

13.6.1 エンコーダ

エンコーダのインターフェースでは、エンコーダが可変長系列を入力 `X` として受け取ることだけを指定する。実装は、この基底 `Encoder` クラスを継承する任意のモデルによって提供される。

```
class Encoder(nn.Module):  #@save
    """The base encoder interface for the encoder--decoder architecture."""
    def __init__(self):
        super().__init__()

    # Later there can be additional arguments (e.g., length excluding padding)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, X, *args):
        raise NotImplementedError
```

```
class Encoder(nn.Block):  #@save
    """The base encoder interface for the encoder--decoder architecture."""
    def __init__(self):
        super().__init__()

    # Later there can be additional arguments (e.g., length excluding padding)
    def forward(self, X, *args):
        raise NotImplementedError
```

```
class Encoder(nn.Module):  #@save
    """The base encoder interface for the encoder--decoder architecture."""
    def setup(self):
        raise NotImplementedError

    # Later there can be additional arguments (e.g., length excluding padding)
    def __call__(self, X, *args):
        raise NotImplementedError
```

```
class Encoder(tf.keras.layers.Layer):  #@save
    """The base encoder interface for the encoder--decoder architecture."""
    def __init__(self):
        super().__init__()

    # Later there can be additional arguments (e.g., length excluding padding)
    def call(self, X, *args):
        raise NotImplementedError
```

13.6.2 デコーダ

以下のデコーダのインターフェースでは、エンコーダの出力 (`enc_all_outputs`) をエンコードされた状態に変換するための追加の `init_state` メソッドを加える。このステップでは、入力の有効長など、追加の入力が必要になる場合があることに注意されたい。これは Section 13.5 で説明した。可変長系列をトークンごとに生成するために、デコーダは毎回、入力（たとえば、1 つ前の時刻ステップで生成されたトークン）とエンコードされた状態を現在の時刻ステップにおける出力トークンへ写像する。

```
class Decoder(nn.Module):  #@save
    """The base decoder interface for the encoder--decoder architecture."""
    def __init__(self):
        super().__init__()

    # Later there can be additional arguments (e.g., length excluding padding)
    def init_state(self, enc_all_outputs, *args):
        raise NotImplementedError
```

(continues on next page)

(continued from previous page)

```
    def forward(self, X, state):
        raise NotImplementedError
```

```
class Decoder(nn.Block):  #@save
    """The base decoder interface for the encoder--decoder architecture."""
    def __init__(self):
        super().__init__()

    # Later there can be additional arguments (e.g., length excluding padding)
    def init_state(self, enc_all_outputs, *args):
        raise NotImplementedError

    def forward(self, X, state):
        raise NotImplementedError
```

```
class Decoder(nn.Module):  #@save
    """The base decoder interface for the encoder--decoder architecture."""
    def setup(self):
        raise NotImplementedError

    # Later there can be additional arguments (e.g., length excluding padding)
    def init_state(self, enc_all_outputs, *args):
        raise NotImplementedError

    def __call__(self, X, state):
        raise NotImplementedError
```

```
class Decoder(tf.keras.layers.Layer):  #@save
    """The base decoder interface for the encoder--decoder architecture."""
    def __init__(self):
        super().__init__()

    # Later there can be additional arguments (e.g., length excluding padding)
    def init_state(self, enc_all_outputs, *args):
        raise NotImplementedError

    def call(self, X, state):
        raise NotImplementedError
```

13.6.3 エンコーダとデコーダを組み合わせる

順伝播では、エンコーダの出力がエンコードされた状態を生成するために使われ、この状態はデコーダによってその入力の 1 つとしてさらに利用される。

```
class EncoderDecoder(d2l.Classifier):  #@save
    """The base class for the encoder--decoder architecture."""
    def __init__(self, encoder, decoder):
```

(continues on next page)

(continued from previous page)

```
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        enc_all_outputs = self.encoder(enc_X, *args)
        dec_state = self.decoder.init_state(enc_all_outputs, *args)
        # Return decoder output only
        return self.decoder(dec_X, dec_state)[0]
```

```
class EncoderDecoder(d2l.Classifier):  #@save
    """The base class for the encoder--decoder architecture."""
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        enc_all_outputs = self.encoder(enc_X, *args)
        dec_state = self.decoder.init_state(enc_all_outputs, *args)
        # Return decoder output only
        return self.decoder(dec_X, dec_state)[0]
```

```
class EncoderDecoder(d2l.Classifier):  #@save
    """The base class for the encoder--decoder architecture."""
    encoder: nn.Module
    decoder: nn.Module
    training: bool

    def __call__(self, enc_X, dec_X, *args):
        enc_all_outputs = self.encoder(enc_X, *args, training=self.training)
        dec_state = self.decoder.init_state(enc_all_outputs, *args)
        # Return decoder output only
        return self.decoder(dec_X, dec_state, training=self.training)[0]
```

```
class EncoderDecoder(d2l.Classifier):  #@save
    """The base class for the encoder--decoder architecture."""
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def call(self, enc_X, dec_X, *args):
        enc_all_outputs = self.encoder(enc_X, *args, training=True)
        dec_state = self.decoder.init_state(enc_all_outputs, *args)
        # Return decoder output only
        return self.decoder(dec_X, dec_state, training=True)[0]
```

次の節では、このエンコーダ–デコーダアーキテクチャに基づいて RNN を用いてシーケンス・ツー・シーケンスモデルを設計する方法を見る。

13.6.4 まとめ

エンコーダ–デコーダアーキテクチャは、入力と出力の両方が可変長系列からなる場合に対応できるため、機械翻訳のようなシーケンス・ツー・シーケンス問題に適している。エンコーダは可変長系列を入力として受け取り、それを固定形状の状態へ変換する。デコーダは、その固定形状のエンコードされた状態を可変長系列へ写像する。

13.6.5 演習

1. ニューラルネットワークを用いてエンコーダ–デコーダアーキテクチャを実装すると仮定する。エンコーダとデコーダは同じ種類のニューラルネットワークでなければならないだろうか。
2. 機械翻訳以外に、エンコーダ–デコーダアーキテクチャを適用できる別の応用例を考えられるか。

13.7 機械翻訳のためのシーケンス・ツー・シーケンス学習

いわゆるシーケンス・ツー・シーケンス問題、たとえば機械翻訳 (Section 13.5 で議論したように) では、入力と出力がそれぞれ長さ可変で対応の取れていない系列から構成されるため、一般にエンコーダ–デコーダアーキテクチャ (Section 13.6) に依拠する。この節では、エンコーダとデコーダの両方を RNN として実装したエンコーダ–デコーダアーキテクチャを、機械翻訳のタスクに適用する例を示す (Cho *et al.*, 2014, Sutskever *et al.*, 2014)。

ここで、エンコーダ RNN は長さ可変の系列を入力として受け取り、それを固定形状の隠れ状態へ変換する。後の Section 14 では、系列全体を単一の固定長表現に圧縮しなくても符号化された入力にアクセスできる注意機構を導入する。

次に、出力系列を 1 トークンずつ生成するために、別個の RNN からなるデコーダモデルが、入力系列と出力側の前のトークンの両方を与えられて、各時刻の次のターゲットトークンを予測する。学習時には、デコーダは通常、公式の「正解」ラベルにおける前のトークンを条件として与えられる。しかしテスト時には、すでに予測済みのトークンを条件としてデコーダの各出力を生成したいと考える。なお、エンコーダを無視すれば、シーケンス・ツー・シーケンスアーキテクチャのデコーダは通常の言語モデルとまったく同じように振る舞いる。Fig. 13.7.1 は、機械翻訳におけるシーケンス・ツー・シーケンス学習で 2 つの RNN をどのように使うかを示している。

Fig. 13.7.1 では、特別な“<eos>”トークンが系列の終端を示す。このトークンが生成されると、モデルは予測を停止できる。RNN デコーダの初期時刻には、注意

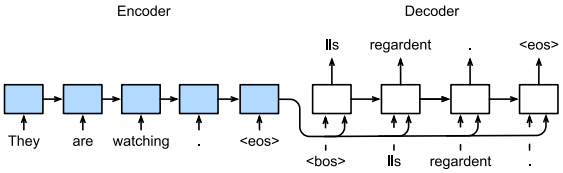


Fig. 13.7.1 RNN エンコーダと RNN デコーダによるシーケンス・ツー・シーケンス学習。

すべき特別な設計上の決定が 2 つある。第 1 に、すべての入力の先頭に特別な系列開始“<bos>”トークンを置きる。第 2 に、エンコーダの最終隠れ状態をデコーダへ各デコード時刻ごとに入力することもできる (Cho *et al.*, 2014)。Sutskever *et al.* (2014) のような別の設計では、RNN エンコーダの最終隠れ状態はデコーダの隠れ状態を初期化するために最初のデコード時刻にのみ用いられる。

```
import collections
from d2l import torch as d2l
import math
import torch
from torch import nn
from torch.nn import functional as F
```

```
import collections
from d2l import mxnet as d2l
import math
from mxnet import np, npx, init, gluon, autograd
from mxnet.gluon import nn, rnn
npx.set_np()
```

```
import collections
from d2l import jax as d2l
from flax import linen as nn
from functools import partial
import jax
from jax import numpy as jnp
import math
import optax
```

```
import collections
from d2l import tensorflow as d2l
import math
import tensorflow as tf
```

13.7.1 Teacher Forcing

入力系列に対してエンコーダを実行するのは比較的単純であるが、デコーダの入力と出力の扱いにはより注意が必要である。最も一般的な方法は、しばしば *teacher forcing* と呼ばれる。ここでは、元のターゲット系列（トークンラベル）をデコーダへの入力として与える。より具体的には、特別な系列開始トークン

と、最後のトークンを除いた元のターゲット系列をデコーダへの入力として連結し、一方でデコーダの出力（学習用ラベル）は、1 トークンずらした元のターゲット系列になる: “<bos>”, “Ils”, “regardent”, “.”→ “Ils”, “regardent”, “.”, “<eos>” (Fig. 13.7.1)。

Section 13.5.3 における実装では、teacher forcing のための学習データを準備した。ここでのトークンのシフトは、Section 12.3 における言語モデルの学習と同様の自己教師あり学習である。別の方法としては、前の時刻での 予測済みトークンを現在のデコーダ入力として与えることもできる。

以下では、Fig. 13.7.1 に示した設計をより詳しく説明する。このモデルは、Section 13.5 で導入した英仏データセット上で機械翻訳のために学習する。

13.7.2 エンコーダ

エンコーダは、長さ可変の入力系列を固定形状の コンテキスト変数 $\mathbf{c}$ に変換することを思い出されたい (Fig. 13.7.1 を参照)。

単一の系列例（バッチサイズ 1）を考える。入力系列が $x_1, \ldots, x_T$ であり、x_t が t^{th} トークンだとする。時刻 t において、RNN は x_t に対応する入力特徴ベクトル $\mathbf{x}_t$ と前時刻の隠れ状態 $\mathbf{h}_{t-1}$ を現在の隠れ状態 $\mathbf{h}_t$ に変換する。RNN の再帰層の変換は、関数 f を用いて次のように表せる。

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}). \tag{13.7.1}$$

一般に、エンコーダはすべての時刻の隠れ状態をカスタマイズされた関数 q を通してコンテキスト変数へ変換する。

$$\mathbf{c} = q(\mathbf{h}_1, \ldots, \mathbf{h}_T). \tag{13.7.2}$$

たとえば、Fig. 13.7.1 では、コンテキスト変数は単に隠れ状態 $\mathbf{h}_T$ であり、入力系列の最後のトークンを処理した後のエンコーダ RNN の表現に対応する。

この例では、エンコーダの設計に単方向 RNN を用いており、隠れ状態はその時刻までの入力部分系列のみに依存する。双方向 RNN を用いてエンコーダを構成することもできる。この場合、隠れ状態はその時刻の前後の部分系列（現在時刻の入力を含む）に依存し、系列全体の情報を符号化する。

それでは、RNN エンコーダを実装しよう。入力系列の各トークンに対する特徴ベクトルを得るために、埋め込み層を用いることに注意されたい。埋め込み層の重みは行列であり、行数は入力語彙のサイズ (vocab_size) に対応し、列数は特徴ベクトルの次元 (embed_size) に対応する。任意の入力トークンのインデックス i に対して、埋め込み層は重み行列の i^{th} 行（0 始まり）を取り出してその特徴ベクトルを返す。ここでは、エンコーダを多層 GRU で実装する。

```
def init_seq2seq(module):  #@save
    """Initialize weights for sequence-to-sequence learning."""
    if type(module) == nn.Linear:
```

(continues on next page)

(continued from previous page)

```
            nn.init.xavier_uniform_(module.weight)
        if type(module) == nn.GRU:
            for param in module._flat_weights_names:
                if "weight" in param:
                    nn.init.xavier_uniform_(module._parameters[param])
```

```
class Seq2SeqEncoder(d2l.Encoder):  #@save
    """The RNN encoder for sequence-to-sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(num_hiddens, num_layers, dropout)
        self.initialize(init.Xavier())

    def forward(self, X, *args):
        # X shape: (batch_size, num_steps)
        embs = self.embedding(d2l.transpose(X))
        # embs shape: (num_steps, batch_size, embed_size)
        outputs, state = self.rnn(embs)
        # outputs shape: (num_steps, batch_size, num_hiddens)
        # state shape: (num_layers, batch_size, num_hiddens)
        return outputs, state
```

```
class Seq2SeqEncoder(d2l.Encoder):  #@save
    """The RNN encoder for sequence-to-sequence learning."""
    vocab_size: int
    embed_size: int
    num_hiddens: int
    num_layers: int
    dropout: float = 0

    def setup(self):
        self.embedding = nn.Embed(self.vocab_size, self.embed_size)
        self.rnn = d2l.GRU(self.num_hiddens, self.num_layers, self.dropout)

    def __call__(self, X, *args, training=False):
        # X shape: (batch_size, num_steps)
        embs = self.embedding(d2l.astype(d2l.transpose(X), d2l.int32))
        # embs shape: (num_steps, batch_size, embed_size)
        outputs, state = self.rnn(embs, training=training)
        # outputs shape: (num_steps, batch_size, num_hiddens)
        # state shape: (num_layers, batch_size, num_hiddens)
        return outputs, state
```

```
class Seq2SeqEncoder(d2l.Encoder):  #@save
    """The RNN encoder for sequence-to-sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = tf.keras.layers.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(num_hiddens, num_layers, dropout)
```

(continues on next page)

(continued from previous page)

```
    def call(self, X, *args):
        # X shape: (batch_size, num_steps)
        embs = self.embedding(d2l.transpose(X))
        # embs shape: (num_steps, batch_size, embed_size)
        outputs, state = self.rnn(embs)
        # outputs shape: (num_steps, batch_size, num_hiddens)
        # state shape: (num_layers, batch_size, num_hiddens)
        return outputs, state
```

```
class Seq2SeqEncoder(d2l.Encoder):  #@save
    """The RNN encoder for sequence-to-sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(embed_size, num_hiddens, num_layers, dropout)
        self.apply(init_seq2seq)

    def forward(self, X, *args):
        # X shape: (batch_size, num_steps)
        embs = self.embedding(d2l.astype(d2l.transpose(X), d2l.int64))
        # embs shape: (num_steps, batch_size, embed_size)
        outputs, state = self.rnn(embs)
        # outputs shape: (num_steps, batch_size, num_hiddens)
        # state shape: (num_layers, batch_size, num_hiddens)
        return outputs, state
```

```
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
batch_size, num_steps = 4, 9
encoder = Seq2SeqEncoder(vocab_size, embed_size, num_hiddens, num_layers)
X = d2l.zeros((batch_size, num_steps))
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    enc_outputs, enc_state = encoder(X)
if tab.selected('jax'):
    (enc_outputs, enc_state), _ = encoder.init_with_output(d2l.get_key(), X)

d2l.check_shape(enc_outputs, (num_steps, batch_size, num_hiddens))
```

```
[07:42:49] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
batch_size, num_steps = 4, 9
encoder = Seq2SeqEncoder(vocab_size, embed_size, num_hiddens, num_layers)
X = d2l.zeros((batch_size, num_steps))
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    enc_outputs, enc_state = encoder(X)
if tab.selected('jax'):
    (enc_outputs, enc_state), _ = encoder.init_with_output(d2l.get_key(), X)
```

(continues on next page)

(continued from previous page)

```
d2l.check_shape(enc_outputs, (num_steps, batch_size, num_hiddens))
```

```
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
batch_size, num_steps = 4, 9
encoder = Seq2SeqEncoder(vocab_size, embed_size, num_hiddens, num_layers)
X = d2l.zeros((batch_size, num_steps))
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    enc_outputs, enc_state = encoder(X)
if tab.selected('jax'):
    (enc_outputs, enc_state), _ = encoder.init_with_output(d2l.get_key(), X)

d2l.check_shape(enc_outputs, (num_steps, batch_size, num_hiddens))
```

具体例を使って、上のエンコーダ実装を説明しよう。以下では、隠れユニット数が 16 の 2 層 GRU エンコーダを生成する。系列入力 `X` のミニバッチ（バッチサイズ = 4、時系列長 = 9）が与えられると、すべての時刻における最終層の隠れ状態（エンコーダの再帰層が返す `enc_outputs`）は、（時系列長、バッチサイズ、隠れユニット数）の形状を持つテンソルになる。

```
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
batch_size, num_steps = 4, 9
encoder = Seq2SeqEncoder(vocab_size, embed_size, num_hiddens, num_layers)
X = d2l.zeros((batch_size, num_steps))
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    enc_outputs, enc_state = encoder(X)
if tab.selected('jax'):
    (enc_outputs, enc_state), _ = encoder.init_with_output(d2l.get_key(), X)

d2l.check_shape(enc_outputs, (num_steps, batch_size, num_hiddens))
```

```
if tab.selected('mxnet', 'pytorch', 'jax'):
    d2l.check_shape(enc_state, (num_layers, batch_size, num_hiddens))
if tab.selected('tensorflow'):
    d2l.check_len(enc_state, num_layers)
    d2l.check_shape(enc_state[0], (batch_size, num_hiddens))
```

```
if tab.selected('mxnet', 'pytorch', 'jax'):
    d2l.check_shape(enc_state, (num_layers, batch_size, num_hiddens))
if tab.selected('tensorflow'):
    d2l.check_len(enc_state, num_layers)
    d2l.check_shape(enc_state[0], (batch_size, num_hiddens))
```

```
if tab.selected('mxnet', 'pytorch', 'jax'):
    d2l.check_shape(enc_state, (num_layers, batch_size, num_hiddens))
if tab.selected('tensorflow'):
    d2l.check_len(enc_state, num_layers)
    d2l.check_shape(enc_state[0], (batch_size, num_hiddens))
```

ここでは GRU を使っているので、最終時刻における多層隠れ状態の形状は（隠れ層数、バッチサイズ、隠れユニット数）である。

```
if tab.selected('mxnet', 'pytorch', 'jax'):
    d2l.check_shape(enc_state, (num_layers, batch_size, num_hiddens))
if tab.selected('tensorflow'):
    d2l.check_len(enc_state, num_layers)
    d2l.check_shape(enc_state[0], (batch_size, num_hiddens))
```

```
class Seq2SeqDecoder(d2l.Decoder):
    """The RNN decoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(num_hiddens, num_layers, dropout)
        self.dense = nn.Dense(vocab_size, flatten=False)
        self.initialize(init.Xavier())

    def init_state(self, enc_all_outputs, *args):
        return enc_all_outputs

    def forward(self, X, state):
        # X shape: (batch_size, num_steps)
        # embs shape: (num_steps, batch_size, embed_size)
        embs = self.embedding(d2l.transpose(X))
        enc_output, hidden_state = state
        # context shape: (batch_size, num_hiddens)
        context = enc_output[-1]
        # Broadcast context to (num_steps, batch_size, num_hiddens)
        context = np.tile(context, (embs.shape[0], 1, 1))
        # Concat at the feature dimension
        embs_and_context = d2l.concat((embs, context), -1)
        outputs, hidden_state = self.rnn(embs_and_context, hidden_state)
        outputs = d2l.swapaxes(self.dense(outputs), 0, 1)
        # outputs shape: (batch_size, num_steps, vocab_size)
        # hidden_state shape: (num_layers, batch_size, num_hiddens)
        return outputs, [enc_output, hidden_state]
```

```
class Seq2SeqDecoder(d2l.Decoder):
    """The RNN decoder for sequence to sequence learning."""
    vocab_size: int
    embed_size: int
    num_hiddens: int
    num_layers: int
    dropout: float = 0

    def setup(self):
        self.embedding = nn.Embed(self.vocab_size, self.embed_size)
        self.rnn = d2l.GRU(self.num_hiddens, self.num_layers, self.dropout)
        self.dense = nn.Dense(self.vocab_size)

    def init_state(self, enc_all_outputs, *args):
        return enc_all_outputs
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, X, state, training=False):
        # X shape: (batch_size, num_steps)
        # embs shape: (num_steps, batch_size, embed_size)
        embs = self.embedding(d2l.astype(d2l.transpose(X), d2l.int32))
        enc_output, hidden_state = state
        # context shape: (batch_size, num_hiddens)
        context = enc_output[-1]
        # Broadcast context to (num_steps, batch_size, num_hiddens)
        context = jnp.tile(context, (embs.shape[0], 1, 1))
        # Concat at the feature dimension
        embs_and_context = d2l.concat((embs, context), -1)
        outputs, hidden_state = self.rnn(embs_and_context, hidden_state,
                                         training=training)
        outputs = d2l.swapaxes(self.dense(outputs), 0, 1)
        # outputs shape: (batch_size, num_steps, vocab_size)
        # hidden_state shape: (num_layers, batch_size, num_hiddens)
        return outputs, [enc_output, hidden_state]
```

```
class Seq2SeqDecoder(d2l.Decoder):
    """The RNN decoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = tf.keras.layers.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(num_hiddens, num_layers, dropout)
        self.dense = tf.keras.layers.Dense(vocab_size)

    def init_state(self, enc_all_outputs, *args):
        return enc_all_outputs

    def call(self, X, state):
        # X shape: (batch_size, num_steps)
        # embs shape: (num_steps, batch_size, embed_size)
        embs = self.embedding(d2l.transpose(X))
        enc_output, hidden_state = state
        # context shape: (batch_size, num_hiddens)
        context = enc_output[-1]
        # Broadcast context to (num_steps, batch_size, num_hiddens)
        context = tf.tile(tf.expand_dims(context, 0), (embs.shape[0], 1, 1))
        # Concat at the feature dimension
        embs_and_context = d2l.concat((embs, context), -1)
        outputs, hidden_state = self.rnn(embs_and_context, hidden_state)
        outputs = d2l.transpose(self.dense(outputs), (1, 0, 2))
        # outputs shape: (batch_size, num_steps, vocab_size)
        # hidden_state shape: (num_layers, batch_size, num_hiddens)
        return outputs, [enc_output, hidden_state]
```

13.7.3 デコーダ

各時刻 t' に対するターゲット出力系列 $y_1, y_2, \ldots, y_{T'}$（入力系列の時刻と区別するために t' を用いる）が与えられると、デコーダは、ターゲット側の前のトークン $y_1, \ldots, y_{t'}$ とコンテキスト変数 $\mathbf{c}$ に条件づけて、ステップ $y_{t'+1}$ に現れうる

各トークンに対する予測確率を割り当てる。すなわち、$P(y_{t'+1} \mid y_1, \ldots, y_{t'}, \mathbf{c})$ である。

ターゲット系列の次のトークン $t'+1$ を予測するために、RNN デコーダは前時刻のターゲットトークン $y_{t'}$、前時刻の RNN 隠れ状態 $\mathbf{s}_{t'-1}$、およびコンテキスト変数 $\mathbf{c}$ を入力として受け取り、それらを現在時刻の隠れ状態 $\mathbf{s}_{t'}$ に変換する。デコーダの隠れ層の変換は、関数 g を用いて次のように表せる。

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}). \tag{13.7.3}$$

デコーダの隠れ状態を得た後、出力層と softmax 演算を用いて、次の出力トークン $t'+1$ に対する予測分布 $p(y_{t'+1} \mid y_1, \ldots, y_{t'}, \mathbf{c})$ を計算できる。

Fig. 13.7.1 に従って、以下のようにデコーダを実装する際には、エンコーダの最終時刻の隠れ状態をそのままデコーダの隠れ状態の初期化に用いる。このため、RNN エンコーダと RNN デコーダは同じ層数と同じ隠れユニット数を持つ必要がある。符号化された入力系列情報をさらに取り込むために、コンテキスト変数はすべての時刻でデコーダ入力に連結される。出力トークンの確率分布を予測するために、RNN デコーダの最終層の隠れ状態を全結合層で変換する。

```
class Seq2SeqDecoder(d2l.Decoder):
    """The RNN decoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(embed_size+num_hiddens, num_hiddens,
                           num_layers, dropout)
        self.dense = nn.LazyLinear(vocab_size)
        self.apply(init_seq2seq)

    def init_state(self, enc_all_outputs, *args):
        return enc_all_outputs

    def forward(self, X, state):
        # X shape: (batch_size, num_steps)
        # embs shape: (num_steps, batch_size, embed_size)
        embs = self.embedding(d2l.astype(d2l.transpose(X), d2l.int32))
        enc_output, hidden_state = state
        # context shape: (batch_size, num_hiddens)
        context = enc_output[-1]
        # Broadcast context to (num_steps, batch_size, num_hiddens)
        context = context.repeat(embs.shape[0], 1, 1)
        # Concat at the feature dimension
        embs_and_context = d2l.concat((embs, context), -1)
        outputs, hidden_state = self.rnn(embs_and_context, hidden_state)
        outputs = d2l.swapaxes(self.dense(outputs), 0, 1)
        # outputs shape: (batch_size, num_steps, vocab_size)
        # hidden_state shape: (num_layers, batch_size, num_hiddens)
        return outputs, [enc_output, hidden_state]
```

```
decoder = Seq2SeqDecoder(vocab_size, embed_size, num_hiddens, num_layers)
if tab.selected('mxnet', 'pytorch', 'tensorflow'):
    state = decoder.init_state(encoder(X))
    dec_outputs, state = decoder(X, state)
if tab.selected('jax'):
    state = decoder.init_state(encoder.init_with_output(d2l.get_key(), X)[0])
    (dec_outputs, state), _ = decoder.init_with_output(d2l.get_key(), X,
                                                       state)


d2l.check_shape(dec_outputs, (batch_size, num_steps, vocab_size))
if tab.selected('mxnet', 'pytorch', 'jax'):
    d2l.check_shape(state[1], (num_layers, batch_size, num_hiddens))
if tab.selected('tensorflow'):
    d2l.check_len(state[1], num_layers)
    d2l.check_shape(state[1][0], (batch_size, num_hiddens))
```


```
decoder = Seq2SeqDecoder(vocab_size, embed_size, num_hiddens, num_layers)
if tab.selected('mxnet', 'pytorch', 'tensorflow'):
    state = decoder.init_state(encoder(X))
    dec_outputs, state = decoder(X, state)
if tab.selected('jax'):
    state = decoder.init_state(encoder.init_with_output(d2l.get_key(), X)[0])
    (dec_outputs, state), _ = decoder.init_with_output(d2l.get_key(), X,
                                                       state)


d2l.check_shape(dec_outputs, (batch_size, num_steps, vocab_size))
if tab.selected('mxnet', 'pytorch', 'jax'):
    d2l.check_shape(state[1], (num_layers, batch_size, num_hiddens))
if tab.selected('tensorflow'):
    d2l.check_len(state[1], num_layers)
    d2l.check_shape(state[1][0], (batch_size, num_hiddens))
```



```
decoder = Seq2SeqDecoder(vocab_size, embed_size, num_hiddens, num_layers)
if tab.selected('mxnet', 'pytorch', 'tensorflow'):
    state = decoder.init_state(encoder(X))
    dec_outputs, state = decoder(X, state)
if tab.selected('jax'):
    state = decoder.init_state(encoder.init_with_output(d2l.get_key(), X)[0])
    (dec_outputs, state), _ = decoder.init_with_output(d2l.get_key(), X,
                                                       state)


d2l.check_shape(dec_outputs, (batch_size, num_steps, vocab_size))
if tab.selected('mxnet', 'pytorch', 'jax'):
    d2l.check_shape(state[1], (num_layers, batch_size, num_hiddens))
if tab.selected('tensorflow'):
    d2l.check_len(state[1], num_layers)
    d2l.check_shape(state[1][0], (batch_size, num_hiddens))
```


実装したデコーダを説明するために、先ほどのエンコーダと同じハイパーパラメータで以下のようにインスタンス化する。ご覧のとおり、デコーダの出力形

状は（バッチサイズ、時系列長、語彙サイズ）となり、テンソルの最後の次元には予測されたトークン分布が格納される。

```
decoder = Seq2SeqDecoder(vocab_size, embed_size, num_hiddens, num_layers)
if tab.selected('mxnet', 'pytorch', 'tensorflow'):
    state = decoder.init_state(encoder(X))
    dec_outputs, state = decoder(X, state)
if tab.selected('jax'):
    state = decoder.init_state(encoder.init_with_output(d2l.get_key(), X)[0])
    (dec_outputs, state), _ = decoder.init_with_output(d2l.get_key(), X,
                                                       state)


d2l.check_shape(dec_outputs, (batch_size, num_steps, vocab_size))
if tab.selected('mxnet', 'pytorch', 'jax'):
    d2l.check_shape(state[1], (num_layers, batch_size, num_hiddens))
if tab.selected('tensorflow'):
    d2l.check_len(state[1], num_layers)
    d2l.check_shape(state[1][0], (batch_size, num_hiddens))
```

```
class Seq2Seq(d2l.EncoderDecoder):  #@save
    """The RNN encoder--decoder for sequence to sequence learning."""
    def __init__(self, encoder, decoder, tgt_pad, lr):
        super().__init__(encoder, decoder)
        self.save_hyperparameters()

    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)

    def configure_optimizers(self):
        # Adam optimizer is used here
        if tab.selected('mxnet'):
            return gluon.Trainer(self.parameters(), 'adam',
                                 {'learning_rate': self.lr})
        if tab.selected('pytorch'):
            return torch.optim.Adam(self.parameters(), lr=self.lr)
        if tab.selected('tensorflow'):
            return tf.keras.optimizers.Adam(learning_rate=self.lr)
```

```
class Seq2Seq(d2l.EncoderDecoder):  #@save
    """The RNN encoder--decoder for sequence to sequence learning."""
    encoder: nn.Module
    decoder: nn.Module
    tgt_pad: int
    lr: float

    def validation_step(self, params, batch, state):
        l, _ = self.loss(params, batch[:-1], batch[-1], state)
        self.plot('loss', l, train=False)

    def configure_optimizers(self):
        # Adam optimizer is used here
        return optax.adam(learning_rate=self.lr)
```

```
class Seq2Seq(d2l.EncoderDecoder):  #@save
    """The RNN encoder--decoder for sequence to sequence learning."""
    def __init__(self, encoder, decoder, tgt_pad, lr):
        super().__init__(encoder, decoder)
        self.save_hyperparameters()

    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)

    def configure_optimizers(self):
        # Adam optimizer is used here
        if tab.selected('mxnet'):
            return gluon.Trainer(self.parameters(), 'adam',
                                 {'learning_rate': self.lr})
        if tab.selected('pytorch'):
            return torch.optim.Adam(self.parameters(), lr=self.lr)
        if tab.selected('tensorflow'):
            return tf.keras.optimizers.Adam(learning_rate=self.lr)
```

上の RNN エンコーダ–デコーダモデルの各層は、Fig. 13.7.2 にまとめられている。

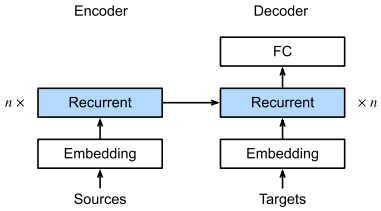


Fig. 13.7.2 RNN エンコーダ–デコーダモデルの層。

13.7.4 シーケンス・ツー・シーケンス学習のためのエンコーダ–デコーダ

これらをすべてコードにまとめると、次のようになる。

```
class Seq2Seq(d2l.EncoderDecoder):  #@save
    """The RNN encoder--decoder for sequence to sequence learning."""
    def __init__(self, encoder, decoder, tgt_pad, lr):
        super().__init__(encoder, decoder)
        self.save_hyperparameters()

    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)

    def configure_optimizers(self):
        # Adam optimizer is used here
        if tab.selected('mxnet'):
```

(continues on next page)

(continued from previous page)

```
            return gluon.Trainer(self.parameters(), 'adam',
                                 {'learning_rate': self.lr})
        if tab.selected('pytorch'):
            return torch.optim.Adam(self.parameters(), lr=self.lr)
        if tab.selected('tensorflow'):
            return tf.keras.optimizers.Adam(learning_rate=self.lr)
```

```
@d2l.add_to_class(Seq2Seq)
def loss(self, Y_hat, Y):
    l = super(Seq2Seq, self).loss(Y_hat, Y, averaged=False)
    mask = d2l.astype(d2l.reshape(Y, -1) != self.tgt_pad, d2l.float32)
    return d2l.reduce_sum(l * mask) / d2l.reduce_sum(mask)
```

```
@d2l.add_to_class(Seq2Seq)
@partial(jax.jit, static_argnums=(0, 5))
def loss(self, params, X, Y, state, averaged=False):
    Y_hat = state.apply_fn({'params': params}, *X,
                           rngs={'dropout': state.dropout_rng})
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    Y = d2l.reshape(Y, (-1,))
    fn = optax.softmax_cross_entropy_with_integer_labels
    l = fn(Y_hat, Y)
    mask = d2l.astype(d2l.reshape(Y, -1) != self.tgt_pad, d2l.float32)
    return d2l.reduce_sum(l * mask) / d2l.reduce_sum(mask), {}
```

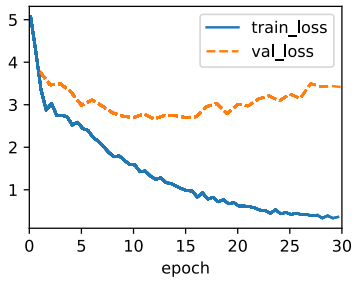
```
@d2l.add_to_class(Seq2Seq)
def loss(self, Y_hat, Y):
    l = super(Seq2Seq, self).loss(Y_hat, Y, averaged=False)
    mask = d2l.astype(d2l.reshape(Y, -1) != self.tgt_pad, d2l.float32)
    return d2l.reduce_sum(l * mask) / d2l.reduce_sum(mask)
```

13.7.5 マスク付き損失関数

各時刻において、デコーダは出力トークンの確率分布を予測する。言語モデルと同様に、softmax を適用して分布を得て、最適化のために交差エントロピー損失を計算できる。Section 13.5 で思い出したように、特別なパディングトークンは系列の末尾に付加されるため、長さの異なる系列を同じ形状のミニバッチとして効率よく読み込める。しかし、パディングトークンの予測は損失計算から除外すべきである。そのために、無関係な要素を 0 でマスクして、無関係な予測に 0 を掛けても結果が 0 になるようにする。

```
@d2l.add_to_class(Seq2Seq)
def loss(self, Y_hat, Y):
    l = super(Seq2Seq, self).loss(Y_hat, Y, averaged=False)
    mask = d2l.astype(d2l.reshape(Y, -1) != self.tgt_pad, d2l.float32)
    return d2l.reduce_sum(l * mask) / d2l.reduce_sum(mask)
```

```
data = d2l.MTFraEng(batch_size=128)
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2
if tab.selected('mxnet', 'pytorch', 'jax'):
    encoder = Seq2SeqEncoder(
        len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
    decoder = Seq2SeqDecoder(
        len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
if tab.selected('mxnet', 'pytorch'):
    model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                    lr=0.005)
if tab.selected('jax'):
    model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                    lr=0.005, training=True)
if tab.selected('mxnet', 'pytorch', 'jax'):
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        encoder = Seq2SeqEncoder(
            len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
        decoder = Seq2SeqDecoder(
            len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
        model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005)
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1)
trainer.fit(model, data)
```

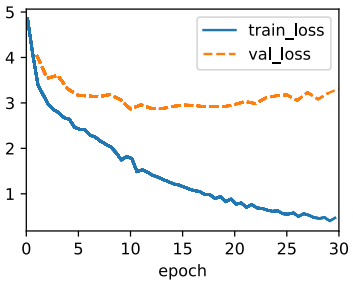



```
data = d2l.MTFraEng(batch_size=128)
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2
if tab.selected('mxnet', 'pytorch', 'jax'):
    encoder = Seq2SeqEncoder(
        len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
    decoder = Seq2SeqDecoder(
        len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
if tab.selected('mxnet', 'pytorch'):
    model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                    lr=0.005)
if tab.selected('jax'):
    model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                    lr=0.005, training=True)
if tab.selected('mxnet', 'pytorch', 'jax'):
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
```

(continues on next page)

(continued from previous page)

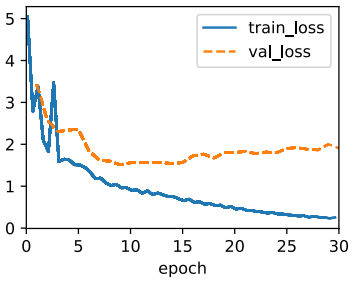
```
    with d2l.try_gpu():
        encoder = Seq2SeqEncoder(
            len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
        decoder = Seq2SeqDecoder(
            len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
        model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005)
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1)
trainer.fit(model, data)
```



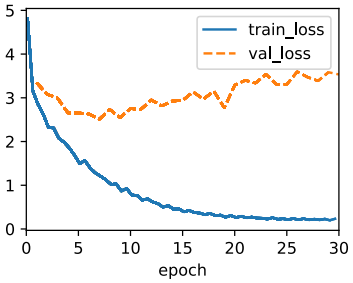

```
data = d2l.MTFraEng(batch_size=128)
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2
if tab.selected('mxnet', 'pytorch', 'jax'):
    encoder = Seq2SeqEncoder(
        len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
    decoder = Seq2SeqDecoder(
        len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
if tab.selected('mxnet', 'pytorch'):
    model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                    lr=0.005)
if tab.selected('jax'):
    model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                    lr=0.005, training=True)
if tab.selected('mxnet', 'pytorch', 'jax'):
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        encoder = Seq2SeqEncoder(
            len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
        decoder = Seq2SeqDecoder(
            len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
        model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005)
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1)
trainer.fit(model, data)
```

13.7.6 学習

これで、機械翻訳データセット上でシーケンス・ツー・シーケンス学習のための RNN エンコーダ–デコーダモデルを作成して学習できる。




```
data = d2l.MTFraEng(batch_size=128)
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2
if tab.selected('mxnet', 'pytorch', 'jax'):
    encoder = Seq2SeqEncoder(
        len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
    decoder = Seq2SeqDecoder(
        len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
if tab.selected('mxnet', 'pytorch'):
    model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                    lr=0.005)
if tab.selected('jax'):
    model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                    lr=0.005, training=True)
if tab.selected('mxnet', 'pytorch', 'jax'):
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        encoder = Seq2SeqEncoder(
            len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
        decoder = Seq2SeqDecoder(
            len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
        model = Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005)
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1)
trainer.fit(model, data)
```




```
@d2l.add_to_class(d2l.EncoderDecoder)  #@save
def predict_step(self, batch, device, num_steps,
                 save_attention_weights=False):
```

(continues on next page)

(continued from previous page)

```
    if tab.selected('mxnet', 'pytorch'):
        batch = [d2l.to(a, device) for a in batch]
    src, tgt, src_valid_len, _ = batch
    if tab.selected('mxnet', 'pytorch'):
        enc_all_outputs = self.encoder(src, src_valid_len)
    if tab.selected('tensorflow'):
        enc_all_outputs = self.encoder(src, src_valid_len, training=False)
    dec_state = self.decoder.init_state(enc_all_outputs, src_valid_len)
    outputs, attention_weights = [d2l.expand_dims(tgt[:, 0], 1), ], []
    for _ in range(num_steps):
        if tab.selected('mxnet', 'pytorch'):
            Y, dec_state = self.decoder(outputs[-1], dec_state)
        if tab.selected('tensorflow'):
            Y, dec_state = self.decoder(outputs[-1], dec_state, training=False)
        outputs.append(d2l.argmax(Y, 2))
        # Save attention weights (to be covered later)
        if save_attention_weights:
            attention_weights.append(self.decoder.attention_weights)
    return d2l.concat(outputs[1:], 1), attention_weights
```

```
@d2l.add_to_class(d2l.EncoderDecoder)  #@save
def predict_step(self, params, batch, num_steps,
                 save_attention_weights=False):
    src, tgt, src_valid_len, _ = batch
    enc_all_outputs, inter_enc_vars = self.encoder.apply(
        {'params': params['encoder']}, src, src_valid_len, training=False,
        mutable='intermediates')
    # Save encoder attention weights if inter_enc_vars containing encoder
    # attention weights is not empty. (to be covered later)
    enc_attention_weights = []
    if bool(inter_enc_vars) and save_attention_weights:
        # Encoder Attention Weights saved in the intermediates collection
        enc_attention_weights = inter_enc_vars[
            'intermediates']['enc_attention_weights'][0]

    dec_state = self.decoder.init_state(enc_all_outputs, src_valid_len)
    outputs, attention_weights = [d2l.expand_dims(tgt[:,0], 1), ], []
    for _ in range(num_steps):
        (Y, dec_state), inter_dec_vars = self.decoder.apply(
            {'params': params['decoder']}, outputs[-1], dec_state,
            training=False, mutable='intermediates')
        outputs.append(d2l.argmax(Y, 2))
        # Save attention weights (to be covered later)
        if save_attention_weights:
            # Decoder Attention Weights saved in the intermediates collection
            dec_attention_weights = inter_dec_vars[
                'intermediates']['dec_attention_weights'][0]
            attention_weights.append(dec_attention_weights)
    return d2l.concat(outputs[1:], 1), (attention_weights,
                                        enc_attention_weights)
```

```
@d2l.add_to_class(d2l.EncoderDecoder)  #@save
def predict_step(self, batch, device, num_steps,
```

(continues on next page)

(continued from previous page)

```
                save_attention_weights=False):
    if tab.selected('mxnet', 'pytorch'):
        batch = [d2l.to(a, device) for a in batch]
    src, tgt, src_valid_len, _ = batch
    if tab.selected('mxnet', 'pytorch'):
        enc_all_outputs = self.encoder(src, src_valid_len)
    if tab.selected('tensorflow'):
        enc_all_outputs = self.encoder(src, src_valid_len, training=False)
    dec_state = self.decoder.init_state(enc_all_outputs, src_valid_len)
    outputs, attention_weights = [d2l.expand_dims(tgt[:, 0], 1), ], []
    for _ in range(num_steps):
        if tab.selected('mxnet', 'pytorch'):
            Y, dec_state = self.decoder(outputs[-1], dec_state)
        if tab.selected('tensorflow'):
            Y, dec_state = self.decoder(outputs[-1], dec_state, training=False)
        outputs.append(d2l.argmax(Y, 2))
        # Save attention weights (to be covered later)
        if save_attention_weights:
            attention_weights.append(self.decoder.attention_weights)
    return d2l.concat(outputs[1:], 1), attention_weights
```

13.7.7 予測

各ステップで出力系列を予測するには、前時刻で予測されたトークンをデコーダへの入力として与える。単純な戦略としては、各ステップの予測時にデコーダが割り当てた最も高い確率を持つトークンを選ぶことである。学習時と同様に、初期時刻には系列開始 (“<bos>”) トークンをデコーダに入力する。この予測過程は Fig. 13.7.3 に示されている。系列終端 (“<eos>”) トークンが予測されたら、出力系列の予測は完了である。

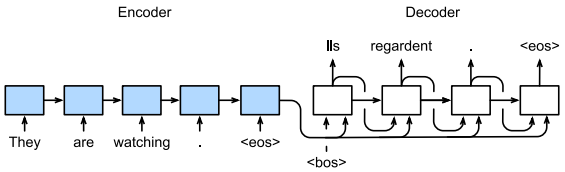


Fig. 13.7.3 RNN エンコーダ–デコーダを用いて出力系列をトークンごとに予測する。

次の節では、ビームサーチに基づくより洗練された戦略を導入する (Section 13.8)。

```
@d2l.add_to_class(d2l.EncoderDecoder)  #@save
def predict_step(self, batch, device, num_steps,
                 save_attention_weights=False):
    if tab.selected('mxnet', 'pytorch'):
        batch = [d2l.to(a, device) for a in batch]
    src, tgt, src_valid_len, _ = batch
    if tab.selected('mxnet', 'pytorch'):
        enc_all_outputs = self.encoder(src, src_valid_len)
```

(continues on next page)

(continued from previous page)

```
    if tab.selected('tensorflow'):
        enc_all_outputs = self.encoder(src, src_valid_len, training=False)
    dec_state = self.decoder.init_state(enc_all_outputs, src_valid_len)
    outputs, attention_weights = [d2l.expand_dims(tgt[:, 0], 1), ], []
    for _ in range(num_steps):
        if tab.selected('mxnet', 'pytorch'):
            Y, dec_state = self.decoder(outputs[-1], dec_state)
        if tab.selected('tensorflow'):
            Y, dec_state = self.decoder(outputs[-1], dec_state, training=False)
        outputs.append(d2l.argmax(Y, 2))
        # Save attention weights (to be covered later)
        if save_attention_weights:
            attention_weights.append(self.decoder.attention_weights)
    return d2l.concat(outputs[1:], 1), attention_weights
```

```
def bleu(pred_seq, label_seq, k):  #@save
    """Compute the BLEU."""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, min(k, len_pred) + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

```
def bleu(pred_seq, label_seq, k):  #@save
    """Compute the BLEU."""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, min(k, len_pred) + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

```
def bleu(pred_seq, label_seq, k):  #@save
    """Compute the BLEU."""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
```

(continues on next page)

(continued from previous page)

```
    for n in range(1, min(k, len_pred) + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

13.7.8 予測系列の評価

予測された系列は、それをターゲット系列（正解）と比較することで評価できる。しかし、2 つの系列の類似度を比較するための適切な尺度は正確には何であろうか。

Bilingual Evaluation Understudy（BLEU）は、もともと機械翻訳結果を評価するために提案されたが (Papineni *et al.*, 2002)、さまざまな応用における出力系列の品質を測るために広く用いられている。原理的には、予測系列中の任意の n-gram (Section 12.3.1) について、BLEU はその n-gram がターゲット系列に現れるかどうかを評価する。

n-gram の適合率を p_n と表す。これは、予測系列とターゲット系列で一致した n-gram の数を予測系列中の n-gram の数で割った比として定義される。説明のために、ターゲット系列が A, B, C, D, E, F で、予測系列が A, B, B, C, D だとすると、$p_1 = 4/5, p_2 = 3/4, p_3 = 1/3, p_4 = 0$ となる。ここで、$\text{len}_{\text{label}}$ と len_{pred} をそれぞれターゲット系列と予測系列のトークン数とする。すると、BLEU は次のように定義される。

$$\exp\left(\min\left(0, 1 - \frac{\text{len}_{\text{label}}}{\text{len}_{\text{pred}}}\right)\right) \prod_{n=1}^{k} p_n^{1/2^n}, \tag{13.7.4}$$

ここで k は一致を調べる最長の n-gram である。

(13.7.4) における BLEU の定義に基づけば、予測系列がターゲット系列と同じであるとき、BLEU は常に 1 になる。さらに、長い n-gram の一致はより難しいため、BLEU は長い n-gram の適合率が高いときにより大きな重みを与える。具体的には、p_n が固定されているとき、$p_n^{1/2^n}$ は n が大きくなるにつれて増加する（元論文では $p_n^{1/n}$ を用いている）。さらに、短い系列を予測すると p_n の値が高くなりやすいため、(13.7.4) の積の前の係数は短い予測系列にペナルティを与える。たとえば、$k = 2$ のとき、ターゲット系列が A, B, C, D, E, F で、予測系列が A, B だとすると、$p_1 = p_2 = 1$ であっても、ペナルティ係数 $\exp(1 - 6/2) \approx 0.14$ により BLEU は低くなる。

BLEU 尺度を次のように実装する。

```
def bleu(pred_seq, label_seq, k):  #@save
    """Compute the BLEU."""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, min(k, len_pred) + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

```
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    preds, _ = model.predict_step(
        data.build(engs, fras), d2l.try_gpu(), data.num_steps)
if tab.selected('jax'):
    preds, _ = model.predict_step(trainer.state.params, data.build(engs, fras),
                                  data.num_steps)
for en, fr, p in zip(engs, fras, preds):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
            break
        translation.append(token)
    print(f'{en} => {translation}, bleu,'
          f'{bleu(" ".join(translation), fr, k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ['j'ai', 'payé', '.'], bleu,0.000
he's calm . => ['il', 'court', '.'], bleu,0.000
i'm home . => ['je', 'suis', 'mouillé', '.'], bleu,0.512
```

```
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    preds, _ = model.predict_step(
        data.build(engs, fras), d2l.try_gpu(), data.num_steps)
if tab.selected('jax'):
    preds, _ = model.predict_step(trainer.state.params, data.build(engs, fras),
                                  data.num_steps)
for en, fr, p in zip(engs, fras, preds):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
            break
        translation.append(token)
```

(continues on next page)

(continued from previous page)

```
    print(f'{en} => {translation}, bleu,'
          f'{bleu(" ".join(translation), fr, k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
he's calm . => ['il', 'est', '<unk>', '.'], bleu,0.658
i'm home . => ['je', 'suis', '<unk>', '.'], bleu,0.512
```

```
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    preds, _ = model.predict_step(
        data.build(engs, fras), d2l.try_gpu(), data.num_steps)
if tab.selected('jax'):
    preds, _ = model.predict_step(trainer.state.params, data.build(engs, fras),
                                  data.num_steps)
for en, fr, p in zip(engs, fras, preds):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
            break
        translation.append(token)
    print(f'{en} => {translation}, bleu,'
          f'{bleu(" ".join(translation), fr, k=2):.3f}')
```

```
go . => ['<unk>', '!'], bleu,0.000
i lost . => ['nous', 'gagné', '.'], bleu,0.000
he's calm . => ['tom', 'est', 'venu', '.'], bleu,0.000
i'm home . => ['je', 'suis', '<unk>', '.'], bleu,0.512
```

最後に、学習済みの RNN エンコーダ–デコーダを用いていくつかの英語文をフランス語に翻訳し、その結果の BLEU を計算する。

```
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    preds, _ = model.predict_step(
        data.build(engs, fras), d2l.try_gpu(), data.num_steps)
if tab.selected('jax'):
    preds, _ = model.predict_step(trainer.state.params, data.build(engs, fras),
                                  data.num_steps)
for en, fr, p in zip(engs, fras, preds):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
            break
        translation.append(token)
    print(f'{en} => {translation}, bleu,'
          f'{bleu(" ".join(translation), fr, k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
he's calm . => ['nous', '<unk>', '.'], bleu,0.000
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

13.7.9 まとめ

エンコーダ–デコーダアーキテクチャの設計に従えば、2 つの RNN を用いてシーケンス・ツー・シーケンス学習のモデルを設計できる。エンコーダ–デコーダの学習では、teacher forcing により予測ではなく元の出力系列をデコーダに入力する。エンコーダとデコーダの実装には、多層 RNN を使える。損失計算などの無関係な計算を除外するために、マスクを使うことができる。出力系列の評価には、BLEU が、予測系列とターゲット系列の間で n-gram を照合する代表的な尺度である。

13.7.10 演習

1. 翻訳結果を改善するためにハイパーパラメータを調整できるか？
2. 損失計算でマスクを使わずに実験を再実行せよ。どのような結果が観察されるか。なぜだろうか。
3. エンコーダとデコーダの層数または隠れユニット数が異なる場合、デコーダの隠れ状態をどのように初期化できるか？
4. 学習時に、teacher forcing の代わりに前時刻の予測をデコーダへ入力するように置き換えよ。性能にどのような影響があるか。
5. GRU を LSTM に置き換えて実験を再実行せよ。
6. デコーダの出力層を設計する他の方法はあるか？

13.8 ビームサーチ

Section 13.7 では、エンコーダ–デコーダアーキテクチャと、それらをエンドツーエンドで学習するための標準的な手法を導入した。しかし、テスト時の予測については、各時刻で次に来ると予測される確率が最も高いトークンを選び、ある時刻で特別な文末トークン“<eos>”を予測するまで続ける *greedy* 戦略だけを述べた。

この節では、まずこの *greedy search* 戦略を形式化し、実務者がしばしば直面する問題を明らかにする。その後、この戦略を 2 つの代替手法、すなわち *exhaustive search*（説明には有用だが実用的ではない）と *beam search*（実務で標準的な手法）と比較する。

まず、Section 13.7 の慣例を借りて、数学的記法を整えよう。任意の時刻 t' において、デコーダは、語彙中の各トークンが次に系列に現れる確率（すなわち $y_{t'+1}$ のもっともらしい値）を表す予測を出力する。これは、直前のトークン $y_1, \ldots, y_{t'}$ と、入力系列を表現するためにエンコーダが生成した文脈変数 $\mathbf{c}$ に条件づけられている。計算コストを定量化するために、出力語彙（特別な文末トークン“<eos>”を含む）を $\mathcal{Y}$ と表す。また、出力系列の最大トークン数を T' とする。私たちの目標は、$O(|\mathcal{Y}|^{T'})$ 個の可能な出力系列の中から、理想的な出力を探索することである。なお、“<eos>”トークンが現れた後はそれ以降のトークンは存在しないため、この数は異なる出力数をやや過大評価している。しかし、ここではこの数が探索空間の大きさをおおよそ捕えていると考えてよい。

13.8.1 Greedy Search

Section 13.7 の単純な *greedy search* 戦略を考える。ここでは、任意の時刻 t' において、$\mathcal{Y}$ の中から条件付き確率が最も高いトークンを単純に選ぶ。すなわち、

$$y_{t'} = \operatorname*{argmax}_{y \in \mathcal{Y}} P(y \mid y_1, \ldots, y_{t'-1}, \mathbf{c}). \tag{13.8.1}$$

モデルが“<eos>”を出力した時点で（あるいは最大長 T' に達した時点で）、出力系列は完了する。

この戦略はもっともらしく見えるし、実際それほど悪くはない。計算負荷が非常に小さいことを考えると、費用対効果はかなり高いと言える。
しかし、効率のことを少し脇に置くと、より合理的なのは、（貪欲に選ばれた）最も確率の高いトークン列ではなく、最も確率の高い系列を探索することだと思えるかもしれない。ところが、この 2 つはかなり異なる場合がある。最も確率の高い系列とは、式 $\prod_{t'=1}^{T'} P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$ を最大化する系列である。機械翻訳の例では、デコーダが本当に基礎となる生成過程の確率を復元しているなら、これが最も確からしい翻訳を与えるはずである。残念ながら、greedy search がこの系列を与える保証はない。

例で見てみよう。出力辞書に“A”、“B”、“C”、“<eos>”の 4 つのトークンがあるとする。Fig. 13.8.1 では、各時刻の下にある 4 つの数は、その時刻に“A”、“B”、“C”、“<eos>”を生成する条件付き確率をそれぞれ表している。

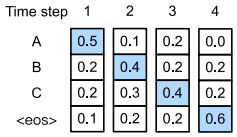


Fig. 13.8.1 各時刻で、greedy search は条件付き確率が最も高いトークンを選ぶ。

各時刻で、greedy search は条件付き確率が最も高いトークンを選ぶ。したがって、出力系列“A”、“B”、“C”、“<eos>”が予測される（Fig. 13.8.1）。この出力系列の条件付き確率は $0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$ である。

次に、Fig. 13.8.2 の別の例を見てみよう。Fig. 13.8.1 とは異なり、時刻 2 では、条件付き確率が *2* 番目に高いトークン“C”を選ぶ。

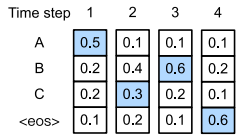


Fig. 13.8.2 各時刻の下にある 4 つの数は、その時刻に“A”、“B”、“C”、“<eos>”を生成する条件付き確率を表す。時刻 2 では、条件付き確率が 2 番目に高いトークン“C”が選ばれる。

時刻 3 は時刻 1 と 2 の出力部分系列に基づくため、その部分系列が Fig. 13.8.1 の“A”と“B”から、Fig. 13.8.2 の“A”と“C”に変わると、Fig. 13.8.2 における時刻 3 の各トークンの条件付き確率も変化する。時刻 3 でトークン“B”を選ぶとしよう。すると時刻 4 は、最初の 3 時刻の出力部分系列“A”、“C”、“B”に条件づけられるが、これは Fig. 13.8.1 の“A”、“B”、“C”とは異なる。したがって、Fig. 13.8.2 における時刻 4 で各トークンを生成する条件付き確率も Fig. 13.8.1 とは異なる。その結果、Fig. 13.8.2 における出力系列“A”、“C”、“B”、“<eos>”の条件付き確率は $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$ となり、Fig. 13.8.1 の greedy search よりも大きくなる。この例では、greedy search によって得られる出力系列“A”、“B”、“C”、“<eos>”は最適ではない。

13.8.2 Exhaustive Search

最も確からしい系列を得ることが目的なら、*exhaustive search* を使うことを考えられる。つまり、あり得るすべての出力系列とその条件付き確率を列挙し、その中で予測確率が最も高いものを出力する。

これなら確かに望む結果が得られるが、計算コストは $O(|\mathcal{Y}|^{T'})$ と非常に高く、系列長に対して指数的であり、しかも語彙サイズという巨大な底を持つ。たとえば、$|\mathcal{Y}| = 10000$、$T' = 10$ のとき、どちらも実用上は小さい数であるが、$10000^{10} = 10^{40}$ 個の系列を評価する必要があり、これは将来見込まれるどんな計算機の能力をも超えている。一方、greedy search の計算コストは $O(|\mathcal{Y}|T')$ である。驚くほど安価であるが、最適とはほど遠いものである。たとえば、$|\mathcal{Y}| = 10000$、$T' = 10$ のとき、評価すべき系列は $10000 \times 10 = 10^5$ 個で済む。

13.8.3 Beam Search

系列デコードの戦略は連続的なスペクトル上にあると考えられる。その中で *beam search* は、greedy search の効率と exhaustive search の最適性の間の妥協点を提供する。最も基本的な beam search は、1 つのハイパーパラメータ、すなわち

beam size k によって特徴づけられる。この用語を説明しよう。時刻 1 では、予測確率が最も高い k 個のトークンを選ぶ。それぞれが、k 個の候補出力系列の最初のトークンになる。以降の各時刻では、前の時刻の k 個の候補出力系列に基づいて、$k|\mathcal{Y}|$ 通りの候補から、予測確率が最も高い k 個の候補出力系列を選び続ける。

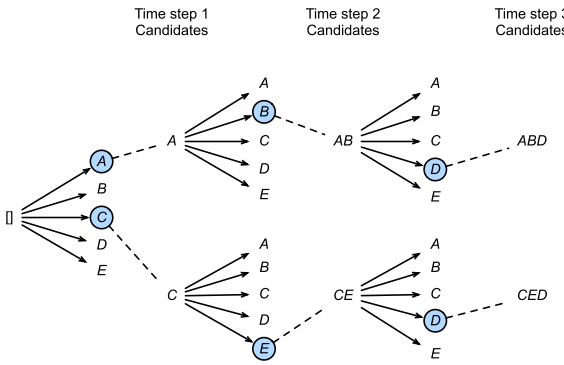


Fig. 13.8.3 ビームサーチの過程（beam size = 2、出力系列の最大長 = 3）。候補出力系列は A、C、AB、CE、ABD、CED である。

Fig. 13.8.3 は、例を用いて beam search の過程を示している。出力語彙が 5 要素、すなわち $\mathcal{Y} = \{A, B, C, D, E\}$ で、そのうち 1 つが“<eos>”だとする。beam size を 2、出力系列の最大長を 3 とする。時刻 1 では、条件付き確率 $P(y_1 \mid \mathbf{c})$ が最も高いトークンが A と C だとしよう。時刻 2 では、すべての $y_2 \in \mathcal{Y}$ について、

$$\begin{aligned} P(A, y_2 \mid \mathbf{c}) &= P(A \mid \mathbf{c})P(y_2 \mid A, \mathbf{c}),\\ P(C, y_2 \mid \mathbf{c}) &= P(C \mid \mathbf{c})P(y_2 \mid C, \mathbf{c}), \end{aligned} \tag{13.8.2}$$

を計算し、これら 10 個の値の中から最大の 2 つ、たとえば $P(A, B \mid \mathbf{c})$ と $P(C, E \mid \mathbf{c})$ を選ぶ。すると時刻 3 では、すべての $y_3 \in \mathcal{Y}$ について、

$$\begin{aligned} P(A, B, y_3 \mid \mathbf{c}) &= P(A, B \mid \mathbf{c})P(y_3 \mid A, B, \mathbf{c}),\\ P(C, E, y_3 \mid \mathbf{c}) &= P(C, E \mid \mathbf{c})P(y_3 \mid C, E, \mathbf{c}), \end{aligned} \tag{13.8.3}$$

を計算し、これら 10 個の値の中から最大の 2 つ、たとえば $P(A, B, D \mid \mathbf{c})$ と $P(C, E, D \mid \mathbf{c})$ を選ぶ。その結果、6 つの候補出力系列が得られる。すなわち、(i) A; (ii) C; (iii) A, B; (iv) C, E; (v) A, B, D; (vi) C, E, D である。

最後に、これら 6 つの系列に基づいて最終的な候補出力系列の集合を得る（たとえば、“<eos>”を含む部分とその後を破棄する）。そして、次のスコアを最大化する出力系列を選ぶ。

$$\frac{1}{L^\alpha} \log P(y_1, \ldots, y_L \mid \mathbf{c}) = \frac{1}{L^\alpha} \sum_{t'=1}^{L} \log P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c}); \tag{13.8.4}$$

ここで L は最終的な候補系列の長さであり、α は通常 0.75 に設定される。

(13.8.4) の和には、長い系列ほど対数項が多く含まれるため、分母の L^{α} は長い系列にペナルティを与える。

beam search の計算コストは $O(k\,|\mathcal{Y}|\,T')$ である。この結果は、greedy search と exhaustive search の中間に位置する。greedy search は、beam size を 1 に設定したときに現れる beam search の特殊な場合とみなせる。

13.8.4 まとめ

系列探索の戦略には、greedy search、exhaustive search、beam search がある。beam search は、beam size を柔軟に選ぶことで、精度と計算コストのトレードオフを提供する。

13.8.5 演習

1. exhaustive search を beam search の特殊な種類として扱うことはできるか。なぜか、あるいはなぜできないか。
2. Section 13.7 の機械翻訳問題に beam search を適用せよ。beam size は翻訳結果と予測速度にどのような影響を与えるか。
3. Section 12.5 では、ユーザが与えたプレフィックスに続くテキストを生成するために言語モデルを使った。これはどの種類の探索戦略を使っているか。改善できるか。

14 注意機構と Transformer

ディープラーニングブームの初期数年間は、主として多層パーセプトロン、畳み込みネットワーク、再帰型ネットワークアーキテクチャを用いて得られた成果によって牽引されていた。驚くべきことに、2010 年代のディープラーニングの多くのブレークスルーを支えたモデルアーキテクチャは、その前身からほぼ 30 年が経過していたにもかかわらず、驚くほど少ししか変わっていなかった。ReLU 活性化関数、残差層、バッチ正規化、ドロップアウト、適応的学習率スケジュールなど、数多くの新しい方法論的革新が多くの実務家のツールキットに加わったが、核となる基盤アーキテクチャは、明らかに古典的なアイデアを大規模化した実装として認識できるものだった。何千もの論文が代替案を提案したにもかかわらず、古典的な畳み込みニューラルネットワーク（Section 10）に似たモデルはコンピュータビジョンにおいて *state-of-the-art* の地位を保ち、Sepp Hochreiter による LSTM 再帰型ニューラルネットワークの元の設計（Section 13.1）に似たモデルは、自然言語処理のほとんどの応用を支配していた。言い換えれば、それまでの時点では、ディープラーニングの急速な台頭は主として利用可能な計算資源の変化（GPU を用いた並列計算の革新のおかげで）と、巨大なデータ資源の利用可能性（安価なストレージとインターネットサービスのおかげで）に起因しているように見えた。これらの要因が今後もこの技術の力を高める主因であり続けるかもしれないが、私たちは今、ついに、支配的なアーキテクチャの景観における大きな転換を目の当たりにしている。

現時点では、ほぼすべての自然言語処理タスクにおける支配的なモデルは Transformer アーキテクチャに基づいている。自然言語処理の新しいタスクが与えられたときの、デフォルトの第一歩は、大規模な Transformer ベースの事前学習済みモデル（たとえば BERT (Devlin *et al.*, 2018)、ELECTRA (Clark *et al.*, 2020)、RoBERTa (Liu *et al.*, 2019)、Longformer (Beltagy *et al.*, 2020)）を取り出し、必要に応じて出力層を適応させ、下流タスクのために利用可能なデータでモデルを微調整することである。ここ数年、OpenAI の大規模言語モデルを中心とした息の詰まるようなニュース報道に注目してきたなら、GPT-2 および GPT-3 の Transformer ベースモデル (Brown *et al.*, 2020, Radford *et al.*, 2019) を中心とする議論を追ってきたことになる。一方、Vision Transformer は、画像認識、物体検出、意味的セグメンテーション、超解像など、多様な視覚タスクに対するデフォルトモデルとして登場した (Dosovitskiy *et al.*, 2021, Liu *et al.*, 2021)。Transformer はまた、音声認識 (Gulati *et al.*, 2020)、強化学習 (Chen *et al.*, 2021)、グラフニューラルネットワーク (Dwivedi and Bresson, 2020) においても競争力のある手法として現れた。

Transformer モデルの核となるアイデアは 注意機構であり、これはもともと、機械翻訳のような系列変換アプリケーションに適用されるエンコーダ–デコーダ

RNN の拡張として構想されたものである (Bahdanau *et al.*, 2014)。機械翻訳の最初期の sequence-to-sequence モデル (Sutskever *et al.*, 2014) では、入力全体がエンコーダによって単一の固定長ベクトルに圧縮され、それがデコーダに渡されていたことを思い出してほしい。注意機構の直感は、入力を圧縮するよりも、デコーダが各ステップで入力系列を再訪するほうがよいかもしれない、というものである。さらに、常に同じ入力表現を見るのではなく、デコーダは特定のデコードステップで入力系列の特定部分に選択的に焦点を当てるべきだと考えられる。Bahdanau の注意機構は、デコーダが各デコードステップで入力の異なる部分に動的に 注意を向けるための簡潔な手段を提供した。高レベルの考え方は、エンコーダが元の入力系列の長さに等しい表現を生成できるというものである。そしてデコード時には、デコーダは（何らかの制御機構を介して）各時刻の入力表現の重み付き和からなるコンテキストベクトルを入力として受け取ることができる。直感的には、重みは各ステップのコンテキストが各入力トークンにどの程度「焦点を当てる」かを決定し、重要なのは、この重み付けの過程を微分可能にして、他のすべてのニューラルネットワークパラメータとともに学習できるようにすることである。

当初、このアイデアは、すでに機械翻訳アプリケーションを支配していた再帰型ニューラルネットワークに対する驚くほど成功した拡張だった。モデルは、元のエンコーダ–デコーダ sequence-to-sequence アーキテクチャよりも良い性能を示した。さらに、注意重みのパターンを調べることで、いくつかの興味深い定性的洞察がしばしば得られることが研究者によって指摘された。翻訳タスクでは、注意モデルはターゲット言語で対応する単語を生成するときに、しばしば言語間の同義語に高い注意重みを割り当てた。たとえば、文“my feet hurt”を“j’ai mal au pieds”に翻訳する際、ニューラルネットワークは対応するフランス語の単語“pieds”を生成するときに“feet”の表現に高い注意重みを割り当てるかもしれない。こうした洞察は、注意モデルが「解釈可能性」をもたらすという主張を後押ししたが、注意重みが正確に何を意味するのか　すなわち、それらをどのように、あるいはそもそも 解釈すべきなのか　は、依然として曖昧な研究テーマである。

しかし、注意機構はやがて、エンコーダ–デコーダ再帰型ニューラルネットワークの拡張としての有用性や、顕著な入力を見つけ出すための有望性を超えて、より重要な関心事として浮上した。Vaswani *et al.* (2017) は機械翻訳のために Transformer アーキテクチャを提案し、再帰接続を完全に排除して、その代わりに巧みに配置された注意機構に依存して入力トークンと出力トークンのあらゆる関係を捉えた。このアーキテクチャは驚くほどうまく機能し、2018 年までに Transformer は大半の state-of-the-art 自然言語処理システムに登場するようになった。さらに同時期に、自然言語処理における支配的な実践は、巨大な汎用コーパス上で大規模モデルを事前学習し、何らかの自己教師あり事前学習目的を最適化したうえで、利用可能な下流データを用いてそれらのモデルを微調整することになった。Transformer と従来アーキテクチャとのギャップは、この事前学習パラダイムで適用したときに特に大きくなり、そのため Transformer の台頭は、現在しばしば *foundation models* と呼ばれるこのような大規模事前学習モデルの台頭と一致していた (Bommasani *et al.*, 2021)。

この章では、最も基本的な直感とこのアイデアの最も単純な実装から始めて、注意モデルを紹介する。その後、Transformer アーキテクチャ、Vision Transformer、そして現代の Transformer ベース事前学習モデルの全体像へと進んでいく。

14.1 クエリ、キー、値

これまでに見てきたネットワークはすべて、入力が明確に定義されたサイズであることに本質的に依存していた。たとえば、ImageNet の画像は 224×224 ピクセルであり、CNN はこのサイズに特化して調整されている。自然言語処理においてさえ、RNN の入力サイズは明確に定義されており固定である。可変サイズへの対応は、1 トークンずつ順に処理するか、あるいは特別に設計された畳み込みカーネルによって行われる (Kalchbrenner *et al.*, 2014)。この方法は、入力が本当に可変サイズであり、しかも情報量もさまざまである場合、たとえば Section 13.7 におけるテキスト変換 (Sutskever *et al.*, 2014) のような場合には、重大な問題を引き起こし得る。特に長い系列では、ネットワークがすでに生成したもの、あるいは見たものすべてを追跡し続けることが非常に難しくなる。Yang *et al.* (2016) が提案したような明示的な追跡ヒューリスティックでさえ、得られる利点は限定的である。

これをデータベースと比べてみよう。最も単純な形では、データベースはキー (k) と値 (v) の集合である。たとえば、私たちのデータベース $\mathcal{D}$ は、{(“Zhang” , “Aston”), (“Lipton”, “Zachary”), (“Li”, “Mu”), (“Smola”, “Alex”), (“Hu”, “Rachel”), (“Werness”, “Brent”)} というタプルからなり、姓がキーで名が値だとする。$\mathcal{D}$ に対して操作を行うことができ、たとえば“Li”に対する厳密なクエリ (q) を与えれば、値“Mu”が返される。もし (“Li”, “Mu”) が $\mathcal{D}$ のレコードでなければ、有効な答えは存在しない。近似一致も許すなら、代わりに (“Lipton”, “Zachary”) を取得することになる。この非常に単純で自明な例であっても、私たちはいくつかの有用なことを学べる。

- データベースサイズに依存せず有効であるように、(k,v) ペアに対して動作するクエリ q を設計できる。
- 同じクエリでも、データベースの内容に応じて異なる答えを返しうる。
- 大きな状態空間（データベース）に対して動作するための「コード」は、かなり単純でよい（たとえば、厳密一致、近似一致、top-k）。
- 操作を有効にするために、データベースを圧縮したり単純化したりする必要はない。

もちろん、ここで単純なデータベースを持ち出したのは、深層学習を説明するためである。実際、これは過去 10 年の深層学習で導入された最も刺激的な概念の一つ、すなわち 注意機構へとつながる (Bahdanau *et al.*, 2014)。その機械翻訳への応用の詳細は後で扱う。ここでは、次のことだけ考えよう。

$\mathcal{D} \stackrel{\text{def}}{=} \{(\mathbf{k}_1, \mathbf{v}_1), \ldots (\mathbf{k}_m, \mathbf{v}_m)\}$ を、キーと 値の m 個のタプルからなるデータベースとする。さらに、$\mathbf{q}$ を クエリとする。このとき、$\mathcal{D}$ 上の *attention* を次のように定義できる。

$$\text{Attention}(\mathbf{q}, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i, \tag{14.1.1}$$

ここで $\alpha(\mathbf{q}, \mathbf{k}_i) \in \mathbb{R}$ $(i = 1, \ldots, m)$ はスカラーの注意重みである。この操作自体は通常、*attention pooling* と呼ばれる。*attention* という名前は、重み α が大きい（すなわち重要な）項に特に注意を払うことに由来する。このように、$\mathcal{D}$ 上の attention は、データベースに含まれる値の線形結合を生成する。実際、これは上の例を、1 つを除くすべての重みが 0 である特殊ケースとして含んでいる。いくつかの特殊ケースがある。

- 重み $\alpha(\mathbf{q}, \mathbf{k}_i)$ が非負である。この場合、注意機構の出力は、値 $\mathbf{v}_i$ が張る凸錐の中に含まれる。
- 重み $\alpha(\mathbf{q}, \mathbf{k}_i)$ が凸結合をなす、すなわち $\sum_i \alpha(\mathbf{q}, \mathbf{k}_i) = 1$ かつすべての i について $\alpha(\mathbf{q}, \mathbf{k}_i) \geq 0$ である。これは深層学習で最も一般的な設定である。
- 重み $\alpha(\mathbf{q}, \mathbf{k}_i)$ のうちちょうど 1 つだけが 1 で、他はすべて 0 である。これは従来のデータベースクエリに似ている。
- すべての重みが等しい、すなわちすべての i について $\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{1}{m}$ である。これはデータベース全体にわたる平均化に相当し、深層学習では average pooling とも呼ばれる。

重みの総和を 1 にするための一般的な戦略は、次のように正規化することである。

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\alpha(\mathbf{q}, \mathbf{k}_i)}{\sum_j \alpha(\mathbf{q}, \mathbf{k}_j)}. \tag{14.1.2}$$

特に、重みを非負にもするためには、指数関数を用いることができる。これは、任意の関数 $a(\mathbf{q}, \mathbf{k})$ を選び、それに対して多項モデルで用いられる softmax 演算を次のように適用できることを意味する。

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_j \exp(a(\mathbf{q}, \mathbf{k}_j))}. \tag{14.1.3}$$

この操作は、すべての深層学習フレームワークで容易に利用できる。これは微分可能であり、その勾配は決して消失しないため、モデルにとって望ましい性質を備えている。ただし、上で導入した注意機構が唯一の選択肢というわけではない。たとえば、強化学習手法を用いて訓練できる非微分可能な注意モデルを設計することもできる (Mnih *et al.*, 2014)。予想されるように、そのようなモデルの訓練はかなり複雑である。そのため、現代の attention 研究の大部分は Fig. 14.1.1 に示した枠組みに従っている。したがって、ここではこの微分可能な機構の系列に焦点を当てる。

非常に注目すべきなのは、キーと値の集合に対して実行する実際の「コード」、すなわちクエリが、操作対象となる空間が大きいにもかかわらず、かな

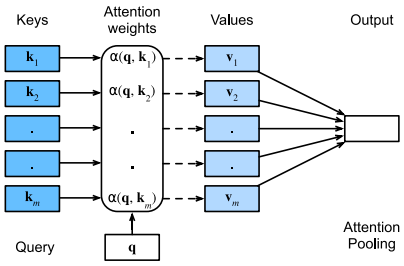


Fig. 14.1.1 注意機構は、クエリ $\mathbf{q}$ とキー $\mathbf{k}_i$ の適合度に基づいて重みを決め、attention pooling を通じて値 $\mathbf{v}_i$ の線形結合を計算する。

り簡潔でありうることである。これは、学習すべきパラメータが多すぎないという点で、ネットワーク層にとって望ましい性質である。同様に便利なのは、attention が、attention pooling のやり方を変えることなく、任意に大きなデータベースに対して動作できることである。

```
from d2l import torch as d2l
import torch
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()
```

```
from d2l import jax as d2l
from jax import numpy as jnp
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

14.1.1 可視化

注意機構の利点の一つは、特に重みが非負で総和が 1 である場合、かなり直感的であることである。この場合、重みが大きいことを、モデルが関連のある成分を選択している方法だと 解釈できるかもしれない。これは良い直感であるが、あくまで 直感にすぎないことを忘れてはいけない。それでも、さまざまなクエリを適用したときに、与えられたキー集合に対してその効果を可視化したいことがある。この関数は後で役立つ。

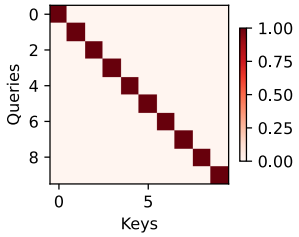
そこで、show_heatmaps 関数を定義する。これは行列（注意重み）を入力として受け取るのではなく、4 つの軸を持つテンソルを受け取り、さまざまなクエリと重みの配列を扱えるようにしている点に注意されたい。したがって、入

力 matrices の形状は（表示する行数、表示する列数、クエリ数、キー数）である。これは、Transformer の仕組みを設計する際にその働きを可視化したいときに役立つ。

```
#@save
def show_heatmaps(matrices, xlabel, ylabel, titles=None, figsize=(2.5, 2.5),
                  cmap='Reds'):
    """Show heatmaps of matrices."""
    d2l.use_svg_display()
    num_rows, num_cols, _, _ = matrices.shape
    fig, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize,
                                 sharex=True, sharey=True, squeeze=False)
    for i, (row_axes, row_matrices) in enumerate(zip(axes, matrices)):
        for j, (ax, matrix) in enumerate(zip(row_axes, row_matrices)):
            if tab.selected('pytorch', 'mxnet', 'tensorflow'):
                pcm = ax.imshow(d2l.numpy(matrix), cmap=cmap)
            if tab.selected('jax'):
                pcm = ax.imshow(matrix, cmap=cmap)
            if i == num_rows - 1:
                ax.set_xlabel(xlabel)
            if j == 0:
                ax.set_ylabel(ylabel)
            if titles:
                ax.set_title(titles[j])
    fig.colorbar(pcm, ax=axes, shrink=0.6);
```

簡単な妥当性確認として、クエリとキーが同じときにのみ注意重みが 1 となる場合を表す単位行列を可視化してみよう。

```
attention_weights = d2l.reshape(d2l.eye(10), (1, 1, 10, 10))
show_heatmaps(attention_weights, xlabel='Keys', ylabel='Queries')
```



14.1.2 まとめ

注意機構により、多数の（キー、値）ペアからデータを集約できる。ここまでの議論はかなり抽象的で、単にデータをプールする方法を説明しているにすぎなかった。これらの不思議なクエリ、キー、値がどこから生じるのかは、まだ説明していない。ここで少し直感を与えると、たとえば回帰の設定では、クエリは回帰を行うべき位置に対応するかもしれない。キーは過去のデータが観測された位置であり、値は（回帰）値そのものである。これは次節で学ぶ、いわゆる Nadaraya–Watson 推定量 (Nadaraya, 1964, Watson, 1964) である。

設計上、注意機構は、ニューラルネットワークが集合から要素を選択し、それに対応する表現の重み付き和を構成するための 微分可能な制御手段を提供する。

14.1.3 演習

1. 古典的なデータベースで使われる近似的な（キー、クエリ）一致を再実装したいとする。どの attention 関数を選ぶか？
2. attention 関数が $a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i$ で与えられ、かつ $i = 1, \ldots, m$ に対して $\mathbf{k}_i = \mathbf{v}_i$ であるとする。(14.1.3) における softmax 正規化を用いたときのキー上の確率分布を $p(\mathbf{k}_i; \mathbf{q})$ と表す。$\nabla_{\mathbf{q}} \text{Attention}(\mathbf{q}, \mathcal{D}) = \text{Cov}_{p(\mathbf{k}_i;\mathbf{q})}[\mathbf{k}_i]$ を証明せよ。
3. 注意機構を用いた微分可能な検索エンジンを設計せよ。
4. Squeeze and Excitation Networks (Hu *et al.*, 2018) の設計を見直し、注意機構の観点から解釈せよ。

14.2 類似度によるアテンションプーリング

ここまででアテンション機構の主要な構成要素を導入したので、今度はそれらをかなり古典的な設定、すなわちカーネル密度推定による回帰と分類 (Nadaraya, 1964, Watson, 1964) に用いてみよう。この寄り道は単に追加の背景を与えるだけである。完全に任意であり、必要なら飛ばして構わない。

Nadaraya–Watson 推定量の本質は、クエリ $\mathbf{q}$ とキー $\mathbf{k}$ を結び付ける何らかの類似度カーネル $\alpha(\mathbf{q}, \mathbf{k})$ にある。代表的なカーネルには次のようなものがある。

$$\begin{aligned} \alpha(\mathbf{q}, \mathbf{k}) &= \exp\left(-\frac{1}{2}\|\mathbf{q} - \mathbf{k}\|^2\right) && \text{Gaussian;} \\ \alpha(\mathbf{q}, \mathbf{k}) &= 1 \text{ if } \|\mathbf{q} - \mathbf{k}\| \leq 1 && \text{Boxcar;} \\ \alpha(\mathbf{q}, \mathbf{k}) &= \max(0, 1 - \|\mathbf{q} - \mathbf{k}\|) && \text{Epanechikov.} \end{aligned} \tag{14.2.1}$$

他にも多くの選択肢がある。より詳しい概説と、カーネルの選択がカーネル密度推定、しばしば *Parzen Windows* とも呼ばれるもの (Parzen, 1957) とどう関係するかについては、Wikipedia の記事[81] を参照されたい。これらのカーネルはいずれもヒューリスティックであり、調整可能である。たとえば、幅は全体としてだけでなく、各座標ごとにも調整できる。いずれにせよ、どれも回帰と分類の両方に対して次の式を導く。

[81]

$$f(\mathbf{q}) = \sum_i \mathbf{v}_i \frac{\alpha(\mathbf{q}, \mathbf{k}_i)}{\sum_j \alpha(\mathbf{q}, \mathbf{k}_j)}. \tag{14.2.2}$$

特徴量とラベルの観測 $(\mathbf{x}_i, y_i)$ を用いる（スカラー）回帰の場合、$\mathbf{v}_i = y_i$ はスカ

ラー、$\mathbf{k}_i = \mathbf{x}_i$ はベクトルであり、クエリ $\mathbf{q}$ は f を評価すべき新しい位置を表す。（多クラス）分類の場合は、y_i の one-hot エンコーディングを用いて $\mathbf{v}_i$ を得る。この推定量の便利な性質の一つは、学習を必要としないことである。さらに、データ量の増加に応じてカーネルを適切に狭めれば、この手法は整合的であり (Mack and Silverman, 1982)、すなわち統計的に最適な解のいずれかに収束する。まずはいくつかのカーネルを見てみよう。

```
%load_ext d2lbook.tab
tab.interact_select('mxnet', 'pytorch', 'tensorflow', 'jax')
```

```
from d2l import torch as d2l
import torch
from torch import nn
from torch.nn import functional as F
import numpy as np

d2l.use_svg_display()
```

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
npx.set_np()
d2l.use_svg_display()
```

```
from d2l import jax as d2l
import jax
from jax import numpy as jnp
from flax import linen as nn
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
import numpy as np

d2l.use_svg_display()
```

14.2.1 カーネルとデータ

この節で定義するすべてのカーネル $\alpha(\mathbf{k}, \mathbf{q})$ は 平行移動および回転に不変である。つまり、$\mathbf{k}$ と $\mathbf{q}$ を同じように平行移動・回転させても、α の値は変わらない。簡単のため、ここではスカラー引数 $k, q \in \mathbb{R}$ を取り、キー $k = 0$ を原点として選ぶ。すると次のようになる。

```
# Define some kernels
def gaussian(x):
    return d2l.exp(-x**2 / 2)

def boxcar(x):
```

(continues on next page)

(continued from previous page)

```
    return d2l.abs(x) < 1.0

def constant(x):
    return 1.0 + 0 * x

if tab.selected('pytorch'):
    def epanechikov(x):
        return torch.max(1 - d2l.abs(x), torch.zeros_like(x))
if tab.selected('mxnet'):
    def epanechikov(x):
        return np.maximum(1 - d2l.abs(x), 0)
if tab.selected('tensorflow'):
    def epanechikov(x):
        return tf.maximum(1 - d2l.abs(x), 0)
if tab.selected('jax'):
    def epanechikov(x):
        return jnp.maximum(1 - d2l.abs(x), 0)
```

```
fig, axes = d2l.plt.subplots(1, 4, sharey=True, figsize=(12, 3))

kernels = (gaussian, boxcar, constant, epanechikov)
names = ('Gaussian', 'Boxcar', 'Constant', 'Epanechikov')
x = d2l.arange(-2.5, 2.5, 0.1)
for kernel, name, ax in zip(kernels, names, axes):
    if tab.selected('pytorch', 'mxnet', 'tensorflow'):
        ax.plot(d2l.numpy(x), d2l.numpy(kernel(x)))
    if tab.selected('jax'):
        ax.plot(x, kernel(x))
    ax.set_xlabel(name)

d2l.plt.show()
```

異なるカーネルは、範囲と滑らかさに関する異なる概念に対応する。たとえば、boxcar カーネルは距離 1（あるいは別に定義したハイパーパラメータ）以内の観測値にしか注目せず、しかもそれを無差別に行う。

Nadaraya–Watson 推定を実際に見てみるために、訓練データを定義しよう。以下では次の依存関係を用いる。

$$y_i = 2\sin(x_i) + x_i + \epsilon, \tag{14.2.3}$$

ここで ϵ は平均 0、分散 1 の正規分布から生成される。40 個の訓練例をサンプルする。

```
def f(x):
    return 2 * d2l.sin(x) + x

n = 40
if tab.selected('pytorch'):
    x_train, _ = torch.sort(d2l.rand(n) * 5)
    y_train = f(x_train) + d2l.randn(n)
```

(continues on next page)

(continued from previous page)

```
if tab.selected('mxnet'):
    x_train = np.sort(d2l.rand(n) * 5, axis=None)
    y_train = f(x_train) + d2l.randn(n)
if tab.selected('tensorflow'):
    x_train = tf.sort(d2l.rand((n,1)) * 5, 0)
    y_train = f(x_train) + d2l.normal((n, 1))
if tab.selected('jax'):
    x_train = jnp.sort(jax.random.uniform(d2l.get_key(), (n,)) * 5)
    y_train = f(x_train) + jax.random.normal(d2l.get_key(), (n,))
x_val = d2l.arange(0, 5, 0.1)
y_val = f(x_val)
```

14.2.2 Nadaraya–Watson 回帰によるアテンションプーリング

データとカーネルがそろったので、あとはカーネル回帰の推定値を計算する関数だけである。なお、簡単な診断を行うために相対的なカーネル重みも得たいので、まず訓練特徴（共変量）`x_train` とすべての検証特徴 `x_val` の間のカーネルを計算する。これにより行列が得られ、それを正規化する。これを訓練ラベル `y_train` と掛け合わせると推定値が得られる。

(14.1.1) のアテンションプーリングを思い出してほしい。各検証特徴をクエリとし、各訓練特徴–ラベルの組をキー–値のペアとみなす。その結果、正規化された相対カーネル重み（以下の `attention_w`）が アテンション重みになる。

```
def nadaraya_watson(x_train, y_train, x_val, kernel):
    dists = d2l.reshape(x_train, (-1, 1)) - d2l.reshape(x_val, (1, -1))
    # Each column/row corresponds to each query/key
    k = d2l.astype(kernel(dists), d2l.float32)
    # Normalization over keys for each query
    attention_w = k / d2l.reduce_sum(k, 0)
    if tab.selected('pytorch'):
        y_hat = y_train@attention_w
    if tab.selected('mxnet'):
        y_hat = np.dot(y_train, attention_w)
    if tab.selected('tensorflow'):
        y_hat = d2l.transpose(d2l.transpose(y_train)@attention_w)
    if tab.selected('jax'):
        y_hat = y_train@attention_w
    return y_hat, attention_w
```

異なるカーネルがどのような推定を生み出すか見てみよう。

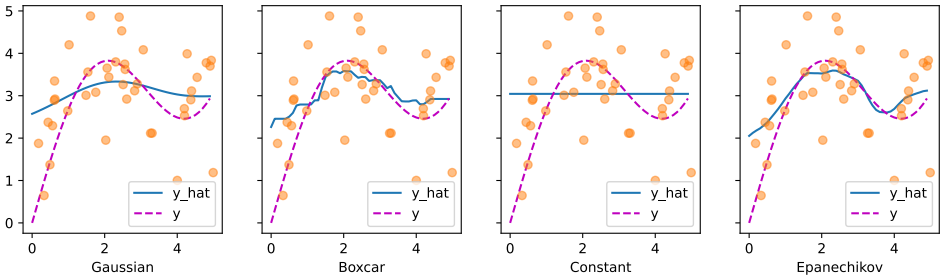
```
def plot(x_train, y_train, x_val, y_val, kernels, names, attention=False):
    fig, axes = d2l.plt.subplots(1, 4, sharey=True, figsize=(12, 3))
    for kernel, name, ax in zip(kernels, names, axes):
        y_hat, attention_w = nadaraya_watson(x_train, y_train, x_val, kernel)
        if attention:
            if tab.selected('pytorch', 'mxnet', 'tensorflow'):
                pcm = ax.imshow(d2l.numpy(attention_w), cmap='Reds')
            if tab.selected('jax'):
```

(continues on next page)

(continued from previous page)

```
                pcm = ax.imshow(attention_w, cmap='Reds')
        else:
            ax.plot(x_val, y_hat)
            ax.plot(x_val, y_val, 'm--')
            ax.plot(x_train, y_train, 'o', alpha=0.5);
        ax.set_xlabel(name)
        if not attention:
            ax.legend(['y_hat', 'y'])
    if attention:
        fig.colorbar(pcm, ax=axes, shrink=0.7)
```

```
plot(x_train, y_train, x_val, y_val, kernels, names)
```



まず目につくのは、3 つの非自明なカーネル（Gaussian、Boxcar、Epanechikov）が、真の関数からそれほど離れていない、かなり実用的な推定を与えていることである。自明な推定 $f(x) = \frac{1}{n}\sum_i y_i$ に帰着する constant カーネルだけが、かなり非現実的な結果を生む。アテンションの重み付けをもう少し詳しく見てみよう。

```
plot(x_train, y_train, x_val, y_val, kernels, names, attention=True)
```

この可視化から、Gaussian、Boxcar、Epanechikov の推定が非常によく似ている理由がはっきり分かる。結局のところ、カーネルの関数形は異なっていても、非常によく似たアテンション重みから導されているからである。ここで疑問になるのは、これが常に成り立つのかということである。

14.2.3 アテンションプーリングの適応

Gaussian カーネルを別の幅のものに置き換えることができる。つまり、$\alpha(\mathbf{q}, \mathbf{k}) = \exp\left(-\frac{1}{2\sigma^2}\|\mathbf{q} - \mathbf{k}\|^2\right)$ を使い、σ^2 でカーネルの幅を決めることができる。これが結果に影響するか見てみよう。

```
sigmas = (0.1, 0.2, 0.5, 1)
names = ['Sigma ' + str(sigma) for sigma in sigmas]
```

(continues on next page)

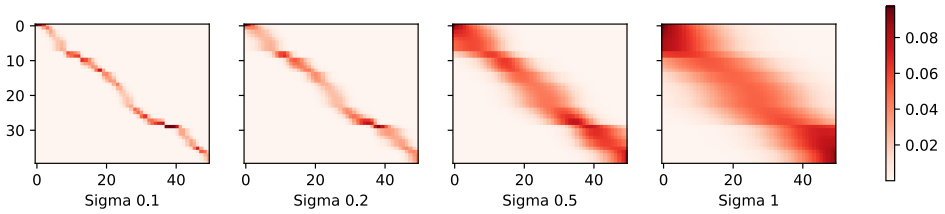
(continued from previous page)

```
def gaussian_with_width(sigma):
    return (lambda x: d2l.exp(-x**2 / (2*sigma**2)))

kernels = [gaussian_with_width(sigma) for sigma in sigmas]
plot(x_train, y_train, x_val, y_val, kernels, names)
```

明らかに、カーネルが狭いほど推定は滑らかでなくなる。同時に、局所的な変動にはよりよく適応する。対応するアテンション重みを見てみよう。

```
plot(x_train, y_train, x_val, y_val, kernels, names, attention=True)
```



予想どおり、カーネルが狭いほど、大きなアテンション重みを持つ範囲も狭くなる。また、同じ幅を選ぶことが必ずしも理想的ではないことも明らかである。実際、Silverman (1986) は局所密度に依存するヒューリスティックを提案した。このような「工夫」は他にも多数提案されている。たとえば、Norelli *et al.* (2022) は、クロスモーダルな画像・テキスト表現を設計するために、類似した最近傍補間技術を用いた。

この方法が半世紀以上前のものであるにもかかわらず、なぜここまで詳しく扱うのか、不思議に思う読者もいるかもしれない。第一に、これは現代のアテンション機構の最も初期の先駆けの一つだからである。第二に、可視化に非常に適している。第三に、そして同じくらい重要であるが、手作りのアテンション機構の限界を示してくれる。より良い戦略は、クエリとキーの表現を学習することで機構そのものを 学習することである。これが次の節で取り組む内容である。

14.2.4 まとめ

Nadaraya–Watson カーネル回帰は、現在のアテンション機構の初期の先駆けである。
分類にも回帰にも、ほとんどあるいはまったく学習や調整を必要とせず、そのまま使える。
アテンション重みは、クエリとキーの類似度（または距離）と、どれだけ類似した観測が利用可能かに応じて割り当てられる。

14.2.5 演習

1. Parzen windows による密度推定は $\hat{p}(\mathbf{x}) = \frac{1}{n}\sum_i k(\mathbf{x}, \mathbf{x}_i)$ で与えられる。二値分類に対して、Parzen windows から得られる関数 $\hat{p}(\mathbf{x}, y = 1) - \hat{p}(\mathbf{x}, y = -1)$ が Nadaraya–Watson 分類と等価であることを証明せよ。

2. Nadaraya–Watson 回帰において、カーネル幅の良い値を学習するための確率的勾配降下法を実装せよ。

   1. 上の推定値をそのまま使って $(f(\mathbf{x_i}) - y_i)^2$ を直接最小化するとどうなるか？ヒント: y_i は f を計算するために使われる項の一部である。

   2. $f(\mathbf{x}_i)$ の推定から $(\mathbf{x}_i, y_i)$ を除外し、カーネル幅について最適化せよ。それでも過学習は観測されるか？

3. すべての $\mathbf{x}$ が単位球面上にある、すなわちすべて $\|\mathbf{x}\| = 1$ を満たすと仮定する。指数関数内の $\|\mathbf{x} - \mathbf{x}_i\|^2$ の項を簡単化できるか？ヒント: これは後で、ドット積アテンションと非常に密接に関係していることが分かる。

4. Mack and Silverman (1982) が Nadaraya–Watson 推定の整合性を証明したことを思い出してほしい。データが増えるにつれて、アテンション機構のスケールをどのくらいの速さで小さくすべきだろうか。答えの直感も述べよ。これはデータの次元に依存するか？どのように依存するか？

14.3 アテンションのスコアリング関数

Section 14.2 では、クエリとキーの相互作用をモデル化するために、ガウスカーネルを含むいくつかの距離ベースのカーネルを用いた。ところが、距離関数は内積よりも計算コストがやや高いことがわかっている。そのため、非負のアテンション重みを保証するソフトマックス演算と組み合わせる場合、計算がより簡単な アテンションのスコアリング関数 a に多くの工夫が注がれてきた。これは (14.1.3) と Fig. 14.3.1 に現れる。

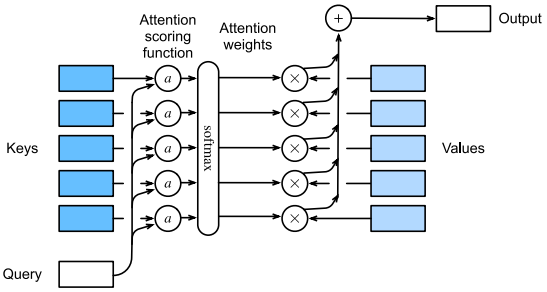


Fig. 14.3.1 アテンションプーリングの出力を値の重み付き平均として計算する。重みはアテンションのスコアリング関数 a とソフトマックス演算で求める。

```
from d2l import torch as d2l
import math
import torch
from torch import nn
```

```
import math
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from jax import numpy as jnp
import jax
import math
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

14.3.1 内積アテンション

まず、ガウスカーネルから得られるアテンション関数（指数関数を除く）を少し見直してみよう。

$$a(\mathbf{q}, \mathbf{k}_i) = -\frac{1}{2}\|\mathbf{q} - \mathbf{k}_i\|^2 = \mathbf{q}^\top \mathbf{k}_i - \frac{1}{2}\|\mathbf{k}_i\|^2 - \frac{1}{2}\|\mathbf{q}\|^2. \tag{14.3.1}$$

まず、最後の項は $\mathbf{q}$ のみに依存することに注意されたい。したがって、すべての $(\mathbf{q}, \mathbf{k}_i)$ の組に対して同じ値である。(14.1.3) で行うように、アテンション重みを 1 に正規化すると、この項は完全に消える。次に、バッチ正規化と層正規化（後で説明する）のどちらも、十分に有界で、しばしば一定のノルム $\|\mathbf{k}_i\|$ を持つ活性化をもたらすことにも注意されたい。たとえば、キー $\mathbf{k}_i$ が層正規化によって生成されている場合がそうである。したがって、結果を大きく変えることなく、この項を a の定義から取り除くことができる。

最後に、指数関数の引数のオーダーを適切に制御する必要がある。クエリ $\mathbf{q} \in \mathbb{R}^d$ とキー $\mathbf{k}_i \in \mathbb{R}^d$ のすべての要素が、平均 0・分散 1 の独立同分布な乱数であると仮定しよう。両ベクトルの内積は平均 0、分散 d になる。ベクトル長に依らず内積の分散を 1 に保つために、スケールド内積アテンションのスコアリング関数を用いる。つまり、内積を $1/\sqrt{d}$ で再スケールする。こうして、Transformer などで使われる最初の一般的なアテンション関数に到達する (Vaswani *et al.*, 2017):

$$a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i / \sqrt{d}. \tag{14.3.2}$$

アテンション重み α は依然として正規化が必要であることに注意されたい。

これを (14.1.3) によりさらに簡単にするため、ソフトマックス演算を用いる。

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \mathrm{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(\mathbf{q}^\top \mathbf{k}_i / \sqrt{d})}{\sum_{j=1} \exp(\mathbf{q}^\top \mathbf{k}_j / \sqrt{d})}. \tag{14.3.3}$$

実際、広く使われているアテンション機構はすべてソフトマックスを用いているため、この章の残りではそれに限定する。

14.3.2 便利な関数

アテンション機構を効率よく実装するために、いくつかの関数が必要である。これには、長さが可変な文字列を扱うためのツール（自然言語処理で一般的）と、ミニバッチ上で効率よく評価するためのツール（バッチ行列積）が含まれる。

マスク付きソフトマックス演算

アテンション機構の最も一般的な応用の 1 つは系列モデルである。したがって、長さの異なる系列を扱える必要がある。場合によっては、そのような系列が同じミニバッチに入ることがあり、短い系列にはダミートークンによるパディングが必要になる（例は Section 13.5 を参照）。これらの特別なトークンは意味を持たない。たとえば、次の 3 つの文があるとする。

```
Dive  into  Deep    Learning
Learn to    code    <blank>
Hello world <blank> <blank>
```

アテンションモデルに空白を入れたくないので、単に $\sum_{i=1}^{n} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$ を、実際の文の長さ $l \leq n$ に応じて $\sum_{i=1}^{l} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$ に制限すればよいのである。これは非常に一般的な問題なので、名前が付いている。それが マスク付きソフトマックス演算である。

実装してみよう。実際の実装では、$i > l$ の $\mathbf{v}_i$ の値をゼロにすることで、わずかにごまかしている。さらに、勾配や値への寄与を実質的に消すために、アテンション重みを -10^6 のような非常に大きな負の値に設定する。これは、線形代数のカーネルや演算子が GPU 向けに強く最適化されており、条件分岐（if then else）を含むコードにするよりも、多少計算を無駄にしても速いからである。

```
def masked_softmax(X, valid_lens):  #@save
    """Perform softmax operation by masking elements on the last axis."""
    # X: 3D tensor, valid_lens: 1D or 2D tensor
    def _sequence_mask(X, valid_len, value=0):
        maxlen = X.size(1)
        mask = torch.arange((maxlen), dtype=torch.float32,
                            device=X.device)[None, :] < valid_len[:, None]
        X[~mask] = value
```

(continues on next page)

(continued from previous page)

```
        return X

    if valid_lens is None:
        return nn.functional.softmax(X, dim=-1)
    else:
        shape = X.shape
        if valid_lens.dim() == 1:
            valid_lens = torch.repeat_interleave(valid_lens, shape[1])
        else:
            valid_lens = valid_lens.reshape(-1)
        # On the last axis, replace masked elements with a very large negative
        # value, whose exponentiation outputs 0
        X = _sequence_mask(X.reshape(-1, shape[-1]), valid_lens, value=-1e6)
        return nn.functional.softmax(X.reshape(shape), dim=-1)
```

```
def masked_softmax(X, valid_lens):  #@save
    """Perform softmax operation by masking elements on the last axis."""
    # X: 3D tensor, valid_lens: 1D or 2D tensor
    if valid_lens is None:
        return npx.softmax(X)
    else:
        shape = X.shape
        if valid_lens.ndim == 1:
            valid_lens = valid_lens.repeat(shape[1])
        else:
            valid_lens = valid_lens.reshape(-1)
        # On the last axis, replace masked elements with a very large negative
        # value, whose exponentiation outputs 0
        X = npx.sequence_mask(X.reshape(-1, shape[-1]), valid_lens, True,
                              value=-1e6, axis=1)
        return npx.softmax(X).reshape(shape)
```

```
def masked_softmax(X, valid_lens):  #@save
    """Perform softmax operation by masking elements on the last axis."""
    # X: 3D tensor, valid_lens: 1D or 2D tensor
    def _sequence_mask(X, valid_len, value=0):
        maxlen = X.shape[1]
        mask = jnp.arange((maxlen),
                          dtype=jnp.float32)[None, :] < valid_len[:, None]
        return jnp.where(mask, X, value)

    if valid_lens is None:
        return nn.softmax(X, axis=-1)
    else:
        shape = X.shape
        if valid_lens.ndim == 1:
            valid_lens = jnp.repeat(valid_lens, shape[1])
        else:
            valid_lens = valid_lens.reshape(-1)
        # On the last axis, replace masked elements with a very large negative
        # value, whose exponentiation outputs 0
        X = _sequence_mask(X.reshape(-1, shape[-1]), valid_lens, value=-1e6)
        return nn.softmax(X.reshape(shape), axis=-1)
```

```
def masked_softmax(X, valid_lens):  #@save
    """Perform softmax operation by masking elements on the last axis."""
    # X: 3D tensor, valid_lens: 1D or 2D tensor
    def _sequence_mask(X, valid_len, value=0):
        maxlen = X.shape[1]
        mask = tf.range(start=0, limit=maxlen, dtype=tf.float32)[
            None, :] < tf.cast(valid_len[:, None], dtype=tf.float32)

        if len(X.shape) == 3:
            return tf.where(tf.expand_dims(mask, axis=-1), X, value)
        else:
            return tf.where(mask, X, value)

    if valid_lens is None:
        return tf.nn.softmax(X, axis=-1)
    else:
        shape = X.shape
        if len(valid_lens.shape) == 1:
            valid_lens = tf.repeat(valid_lens, repeats=shape[1])

        else:
            valid_lens = tf.reshape(valid_lens, shape=-1)
        # On the last axis, replace masked elements with a very large negative
        # value, whose exponentiation outputs 0
        X = _sequence_mask(tf.reshape(X, shape=(-1, shape[-1])), valid_lens,
                           value=-1e6)
        return tf.nn.softmax(tf.reshape(X, shape=shape), axis=-1)
```

この関数がどのように動作するかを示すため、サイズが 2×4 の 2 つの例からなるミニバッチを考え、それぞれの有効長が 2 と 3 であるとする。マスク付きソフトマックス演算の結果、各ベクトルの組について有効長を超える値はすべてゼロとしてマスクされる。

```
masked_softmax(torch.rand(2, 2, 4), torch.tensor([2, 3]))
```

```
tensor([[[0.4639, 0.5361, 0.0000, 0.0000],
         [0.3481, 0.6519, 0.0000, 0.0000]],

        [[0.4261, 0.1981, 0.3759, 0.0000],
         [0.4635, 0.2937, 0.2428, 0.0000]]])
```

```
masked_softmax(np.random.uniform(size=(2, 2, 4)), d2l.tensor([2, 3]))
```

```
masked_softmax(jax.random.uniform(d2l.get_key(), (2, 2, 4)), jnp.array([2, 3]))
```

```
Array([[[0.54144967, 0.4585504 , 0.        , 0.        ],
        [0.54905266, 0.45094734, 0.        , 0.        ]],
```

(continues on next page)

(continued from previous page)

```
       [[0.23869859, 0.52262574, 0.2386757 , 0.        ],
        [0.4002735 , 0.3469047 , 0.25282183, 0.        ]]], dtype=float32)
```

```
masked_softmax(tf.random.uniform(shape=(2, 2, 4)), tf.constant([2, 3]))
```

```
<tf.Tensor: shape=(2, 2, 4), dtype=float32, numpy=
array([[[0.43959942, 0.56040066, 0.        , 0.        ],
        [0.56975996, 0.43024006, 0.        , 0.        ]],

       [[0.3823931 , 0.305755  , 0.3118519 , 0.        ],
        [0.37867242, 0.25606048, 0.36526713, 0.        ]]], dtype=float32)>
```

各例の 2 つのベクトルそれぞれに対して有効長をより細かく指定したい場合は、単に 2 次元の有効長テンソルを使う。すると次のようになる。

```
masked_softmax(torch.rand(2, 2, 4), d2l.tensor([[1, 3], [2, 4]]))
```

```
masked_softmax(np.random.uniform(size=(2, 2, 4)),
               d2l.tensor([[1, 3], [2, 4]]))
```

```
masked_softmax(jax.random.uniform(d2l.get_key(), (2, 2, 4)),
               jnp.array([[1, 3], [2, 4]]))
```

```
Array([[[1.        , 0.        , 0.        , 0.        ],
        [0.29248205, 0.37135315, 0.3361648 , 0.        ]],

       [[0.38551602, 0.61448395, 0.        , 0.        ],
        [0.17129263, 0.21547154, 0.25191936, 0.36131644]]], dtype=float32)
```

```
masked_softmax(tf.random.uniform((2, 2, 4)), tf.constant([[1, 3], [2, 4]]))
```

```
<tf.Tensor: shape=(2, 2, 4), dtype=float32, numpy=
array([[[1.        , 0.        , 0.        , 0.        ],
        [0.4731181 , 0.3053249 , 0.221557  , 0.        ]],

       [[0.3913298 , 0.6086702 , 0.        , 0.        ],
        [0.16784328, 0.24103056, 0.23714985, 0.35397628]]], dtype=float32)>
```

バッチ行列積

もう 1 つよく使われる演算は、行列のバッチ同士を掛け合わせることである。これは、クエリ、キー、値のミニバッチを扱うときに便利である。より具体的

には、次を仮定しよう。

$$\begin{aligned}\mathbf{Q} &= [\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_n] \in \mathbb{R}^{n\times a\times b},\\ \mathbf{K} &= [\mathbf{K}_1, \mathbf{K}_2, \ldots, \mathbf{K}_n] \in \mathbb{R}^{n\times b\times c}.\end{aligned} \tag{14.3.4}$$

このとき、バッチ行列積（BMM）は要素ごとの積を計算する。

$$\text{BMM}(\mathbf{Q}, \mathbf{K}) = [\mathbf{Q}_1\mathbf{K}_1, \mathbf{Q}_2\mathbf{K}_2, \ldots, \mathbf{Q}_n\mathbf{K}_n] \in \mathbb{R}^{n\times a\times c}. \tag{14.3.5}$$

深層学習フレームワークでこれを見てみよう。

```
Q = d2l.ones((2, 3, 4))
K = d2l.ones((2, 4, 6))
d2l.check_shape(torch.bmm(Q, K), (2, 3, 6))
```

```
Q = d2l.ones((2, 3, 4))
K = d2l.ones((2, 4, 6))
d2l.check_shape(npx.batch_dot(Q, K), (2, 3, 6))
```

```
Q = d2l.ones((2, 3, 4))
K = d2l.ones((2, 4, 6))
d2l.check_shape(jax.lax.batch_matmul(Q, K), (2, 3, 6))
```

```
Q = d2l.ones((2, 3, 4))
K = d2l.ones((2, 4, 6))
d2l.check_shape(tf.matmul(Q, K).numpy(), (2, 3, 6))
```

14.3.3 スケールド内積アテンション

(14.3.2) で導入した内積アテンションに戻ろう。一般に、クエリとキーの両方が同じベクトル長、たとえば d を持つことが必要であるが、これは $\mathbf{q}^\top\mathbf{k}$ を $\mathbf{q}^\top\mathbf{M}\mathbf{k}$ に置き換え、$\mathbf{M}$ を両空間を変換するために適切に選んだ行列とすることで簡単に対処できる。ここでは、次元が一致していると仮定する。

実際には、効率のためにミニバッチを考えることが多く、たとえば n 個のクエリと m 個のキー・値ペアに対してアテンションを計算する。このとき、クエリとキーの長さは d、値の長さは v である。したがって、クエリ $\mathbf{Q} \in \mathbb{R}^{n\times d}$、キー $\mathbf{K} \in \mathbb{R}^{m\times d}$、値 $\mathbf{V} \in \mathbb{R}^{m\times v}$ に対するスケールド内積アテンションは次のように書ける。

$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n\times v}. \tag{14.3.6}$$

これをミニバッチに適用する際には、(14.3.5) で導入したバッチ行列積が必要になることに注意されたい。以下のスケールド内積アテンションの実装では、モデル正則化のためにドロップアウトを使う。

```
class DotProductAttention(nn.Module):  #@save
    """Scaled dot product attention."""
    def __init__(self, dropout):
        super().__init__()
        self.dropout = nn.Dropout(dropout)

    # Shape of queries: (batch_size, no. of queries, d)
    # Shape of keys: (batch_size, no. of key-value pairs, d)
    # Shape of values: (batch_size, no. of key-value pairs, value dimension)
    # Shape of valid_lens: (batch_size,) or (batch_size, no. of queries)
    def forward(self, queries, keys, values, valid_lens=None):
        d = queries.shape[-1]
        # Swap the last two dimensions of keys with keys.transpose(1, 2)
        scores = torch.bmm(queries, keys.transpose(1, 2)) / math.sqrt(d)
        self.attention_weights = masked_softmax(scores, valid_lens)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

```
class DotProductAttention(nn.Block):  #@save
    """Scaled dot product attention."""
    def __init__(self, dropout):
        super().__init__()
        self.dropout = nn.Dropout(dropout)

    # Shape of queries: (batch_size, no. of queries, d)
    # Shape of keys: (batch_size, no. of key-value pairs, d)
    # Shape of values: (batch_size, no. of key-value pairs, value dimension)
    # Shape of valid_lens: (batch_size,) or (batch_size, no. of queries)
    def forward(self, queries, keys, values, valid_lens=None):
        d = queries.shape[-1]
        # Set transpose_b=True to swap the last two dimensions of keys
        scores = npx.batch_dot(queries, keys, transpose_b=True) / math.sqrt(d)
        self.attention_weights = masked_softmax(scores, valid_lens)
        return npx.batch_dot(self.dropout(self.attention_weights), values)
```

```
class DotProductAttention(nn.Module):  #@save
    """Scaled dot product attention."""
    dropout: float

    # Shape of queries: (batch_size, no. of queries, d)
    # Shape of keys: (batch_size, no. of key-value pairs, d)
    # Shape of values: (batch_size, no. of key-value pairs, value dimension)
    # Shape of valid_lens: (batch_size,) or (batch_size, no. of queries)
    @nn.compact
    def __call__(self, queries, keys, values, valid_lens=None,
                 training=False):
        d = queries.shape[-1]
        # Swap the last two dimensions of keys with keys.swapaxes(1, 2)
        scores = queries@(keys.swapaxes(1, 2)) / math.sqrt(d)
        attention_weights = masked_softmax(scores, valid_lens)
        dropout_layer = nn.Dropout(self.dropout, deterministic=not training)
        return dropout_layer(attention_weights)@values, attention_weights
```

```
class DotProductAttention(tf.keras.layers.Layer):  #@save
    """Scaled dot product attention."""
    def __init__(self, dropout):
        super().__init__()
        self.dropout = tf.keras.layers.Dropout(dropout)

    # Shape of queries: (batch_size, no. of queries, d)
    # Shape of keys: (batch_size, no. of key-value pairs, d)
    # Shape of values: (batch_size, no. of key-value pairs, value dimension)
    # Shape of valid_lens: (batch_size,) or (batch_size, no. of queries)
    def call(self, queries, keys, values, valid_lens=None, **kwargs):
        d = queries.shape[-1]
        scores = tf.matmul(queries, keys, transpose_b=True)/tf.math.sqrt(
            tf.cast(d, dtype=tf.float32))
        self.attention_weights = masked_softmax(scores, valid_lens)
        return tf.matmul(self.dropout(self.attention_weights, **kwargs),␣
↪values)
```

DotProductAttention クラスがどのように動作するかを示すために、先ほどの加法アテンションの玩具例と同じキー、値、有効長を使う。この例では、ミニバッチサイズを 2、キーと値の総数を 10、値の次元を 4 と仮定する。さらに、各観測の有効長はそれぞれ 2 と 6 とする。すると、出力は $2 \times 1 \times 4$ のテンソル、つまりミニバッチの各例につき 1 行になるはずである。

```
queries = d2l.normal(0, 1, (2, 1, 2))
keys = d2l.normal(0, 1, (2, 10, 2))
values = d2l.normal(0, 1, (2, 10, 4))
valid_lens = d2l.tensor([2, 6])

attention = DotProductAttention(dropout=0.5)
attention.eval()
d2l.check_shape(attention(queries, keys, values, valid_lens), (2, 1, 4))
```

```
queries = d2l.normal(0, 1, (2, 1, 2))
keys = d2l.normal(0, 1, (2, 10, 2))
values = d2l.normal(0, 1, (2, 10, 4))
valid_lens = d2l.tensor([2, 6])

attention = DotProductAttention(dropout=0.5)
attention.initialize()
d2l.check_shape(attention(queries, keys, values, valid_lens), (2, 1, 4))
```

```
queries = jax.random.normal(d2l.get_key(), (2, 1, 2))
keys = jax.random.normal(d2l.get_key(), (2, 10, 2))
values = jax.random.normal(d2l.get_key(), (2, 10, 4))
valid_lens = d2l.tensor([2, 6])

attention = DotProductAttention(dropout=0.5)
(output, attention_weights), params = attention.init_with_output(
    d2l.get_key(), queries, keys, values, valid_lens)
print(output)
```

```
queries = tf.random.normal(shape=(2, 1, 2))
keys = tf.random.normal(shape=(2, 10, 2))
values = tf.random.normal(shape=(2, 10, 4))
valid_lens = tf.constant([2, 6])

attention = DotProductAttention(dropout=0.5)
d2l.check_shape(attention(queries, keys, values, valid_lens, training=False),
                (2, 1, 4))
```

アテンション重みが実際に、それぞれ第 2 列と第 6 列を超える部分で消えているか確認してみよう（有効長を 2 と 6 に設定しているためである）。

```
d2l.show_heatmaps(d2l.reshape(attention.attention_weights, (1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```

```
d2l.show_heatmaps(d2l.reshape(attention.attention_weights, (1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```

```
d2l.show_heatmaps(d2l.reshape(attention_weights, (1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```

```
d2l.show_heatmaps(d2l.reshape(attention.attention_weights, (1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```

14.3.4 加法アテンション

クエリ $\mathbf{q}$ とキー $\mathbf{k}$ が異なる次元のベクトルであるとき、$\mathbf{q}^\top \mathbf{M}\mathbf{k}$ によって次元の不一致を行列で調整する方法もあれば、スコアリング関数として加法アテンションを使う方法もある。もう 1 つの利点は、その名の通りアテンションが加法的であることである。これにより、わずかな計算量の節約が可能になる。クエリ $\mathbf{q} \in \mathbb{R}^q$ とキー $\mathbf{k} \in \mathbb{R}^k$ に対して、加法アテンションのスコアリング関数 (Bahdanau *et al.*, 2014) は次のように与えられる。

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q \mathbf{q} + \mathbf{W}_k \mathbf{k}) \in \mathbb{R}, \tag{14.3.7}$$

ここで、$\mathbf{W}_q \in \mathbb{R}^{h \times q}$、$\mathbf{W}_k \in \mathbb{R}^{h \times k}$、$\mathbf{w}_v \in \mathbb{R}^h$ は学習可能なパラメータである。この項をソフトマックスに入力して、非負性と正規化の両方を保証する。(14.3.7) の同値な解釈として、クエリとキーを連結し、1 つの隠れ層を持つ MLP に入力しているとみなすこともできる。活性化関数として tanh を使い、バイアス項を無効にして、加法アテンションを次のように実装する。

```
class AdditiveAttention(nn.Module):  #@save
    """Additive attention."""
    def __init__(self, num_hiddens, dropout, **kwargs):
        super(AdditiveAttention, self).__init__(**kwargs)
        self.W_k = nn.LazyLinear(num_hiddens, bias=False)
```

(continues on next page)

(continued from previous page)

```
        self.W_q = nn.LazyLinear(num_hiddens, bias=False)
        self.w_v = nn.LazyLinear(1, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, queries, keys, values, valid_lens):
        queries, keys = self.W_q(queries), self.W_k(keys)
        # After dimension expansion, shape of queries: (batch_size, no. of
        # queries, 1, num_hiddens) and shape of keys: (batch_size, 1, no. of
        # key-value pairs, num_hiddens). Sum them up with broadcasting
        features = queries.unsqueeze(2) + keys.unsqueeze(1)
        features = torch.tanh(features)
        # There is only one output of self.w_v, so we remove the last
        # one-dimensional entry from the shape. Shape of scores: (batch_size,
        # no. of queries, no. of key-value pairs)
        scores = self.w_v(features).squeeze(-1)
        self.attention_weights = masked_softmax(scores, valid_lens)
        # Shape of values: (batch_size, no. of key-value pairs, value
        # dimension)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

```
class AdditiveAttention(nn.Block):  #@save
    """Additive attention."""
    def __init__(self, num_hiddens, dropout, **kwargs):
        super(AdditiveAttention, self).__init__(**kwargs)
        # Use flatten=False to only transform the last axis so that the
        # shapes for the other axes are kept the same
        self.W_k = nn.Dense(num_hiddens, use_bias=False, flatten=False)
        self.W_q = nn.Dense(num_hiddens, use_bias=False, flatten=False)
        self.w_v = nn.Dense(1, use_bias=False, flatten=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, queries, keys, values, valid_lens):
        queries, keys = self.W_q(queries), self.W_k(keys)
        # After dimension expansion, shape of queries: (batch_size, no. of
        # queries, 1, num_hiddens) and shape of keys: (batch_size, 1,
        # no. of key-value pairs, num_hiddens). Sum them up with
        # broadcasting
        features = np.expand_dims(queries, axis=2) + np.expand_dims(
            keys, axis=1)
        features = np.tanh(features)
        # There is only one output of self.w_v, so we remove the last
        # one-dimensional entry from the shape. Shape of scores:
        # (batch_size, no. of queries, no. of key-value pairs)
        scores = np.squeeze(self.w_v(features), axis=-1)
        self.attention_weights = masked_softmax(scores, valid_lens)
        # Shape of values: (batch_size, no. of key-value pairs, value
        # dimension)
        return npx.batch_dot(self.dropout(self.attention_weights), values)
```

```
class AdditiveAttention(nn.Module):  #@save
    num_hiddens: int
    dropout: float
```

(continues on next page)

(continued from previous page)

```
    def setup(self):
        self.W_k = nn.Dense(self.num_hiddens, use_bias=False)
        self.W_q = nn.Dense(self.num_hiddens, use_bias=False)
        self.w_v = nn.Dense(1, use_bias=False)

    @nn.compact
    def __call__(self, queries, keys, values, valid_lens, training=False):
        queries, keys = self.W_q(queries), self.W_k(keys)
        # After dimension expansion, shape of queries: (batch_size, no. of
        # queries, 1, num_hiddens) and shape of keys: (batch_size, 1, no. of
        # key-value pairs, num_hiddens). Sum them up with broadcasting
        features = jnp.expand_dims(queries, axis=2) + jnp.expand_dims(keys,␣
↪axis=1)
        features = nn.tanh(features)
        # There is only one output of self.w_v, so we remove the last
        # one-dimensional entry from the shape. Shape of scores: (batch_size,
        # no. of queries, no. of key-value pairs)
        scores = self.w_v(features).squeeze(-1)
        attention_weights = masked_softmax(scores, valid_lens)
        dropout_layer = nn.Dropout(self.dropout, deterministic=not training)
        # Shape of values: (batch_size, no. of key-value pairs, value
        # dimension)
        return dropout_layer(attention_weights)@values, attention_weights
```

```
class AdditiveAttention(tf.keras.layers.Layer):  #@save
    """Additive attention."""
    def __init__(self, key_size, query_size, num_hiddens, dropout, **kwargs):
        super().__init__(**kwargs)
        self.W_k = tf.keras.layers.Dense(num_hiddens, use_bias=False)
        self.W_q = tf.keras.layers.Dense(num_hiddens, use_bias=False)
        self.w_v = tf.keras.layers.Dense(1, use_bias=False)
        self.dropout = tf.keras.layers.Dropout(dropout)

    def call(self, queries, keys, values, valid_lens, **kwargs):
        queries, keys = self.W_q(queries), self.W_k(keys)
        # After dimension expansion, shape of queries: (batch_size, no. of
        # queries, 1, num_hiddens) and shape of keys: (batch_size, 1, no. of
        # key-value pairs, num_hiddens). Sum them up with broadcasting
        features = tf.expand_dims(queries, axis=2) + tf.expand_dims(
            keys, axis=1)
        features = tf.nn.tanh(features)
        # There is only one output of self.w_v, so we remove the last
        # one-dimensional entry from the shape. Shape of scores: (batch_size,
        # no. of queries, no. of key-value pairs)
        scores = tf.squeeze(self.w_v(features), axis=-1)
        self.attention_weights = masked_softmax(scores, valid_lens)
        # Shape of values: (batch_size, no. of key-value pairs, value
        # dimension)
        return tf.matmul(self.dropout(
            self.attention_weights, **kwargs), values)
```

`AdditiveAttention` がどのように動作するかを見てみよう。玩具例では、クエリ、キー、値のサイズをそれぞれ $(2, 1, 20)$、$(2, 10, 2)$、$(2, 10, 4)$ とする。これは `DotProductAttention` のときの選択と同じであるが、今回はクエリが 20 次

元である点が異なる。同様に、ミニバッチ内の系列の有効長として $(2, 6)$ を選ぶ。

```
queries = d2l.normal(0, 1, (2, 1, 20))

attention = AdditiveAttention(num_hiddens=8, dropout=0.1)
attention.eval()
d2l.check_shape(attention(queries, keys, values, valid_lens), (2, 1, 4))
```

```
queries = d2l.normal(0, 1, (2, 1, 20))

attention = AdditiveAttention(num_hiddens=8, dropout=0.1)
attention.initialize()
d2l.check_shape(attention(queries, keys, values, valid_lens), (2, 1, 4))
```

```
queries = jax.random.normal(d2l.get_key(), (2, 1, 20))
attention = AdditiveAttention(num_hiddens=8, dropout=0.1)
(output, attention_weights), params = attention.init_with_output(
    d2l.get_key(), queries, keys, values, valid_lens)
print(output)
```

```
[[[-0.80551296 -0.46030337 -0.02557675  1.0253853 ]]

 [[ 0.25692716  0.2782622  -0.0186431   0.5796091 ]]]
```

```
queries = tf.random.normal(shape=(2, 1, 20))

attention = AdditiveAttention(key_size=2, query_size=20, num_hiddens=8,
                              dropout=0.1)
d2l.check_shape(attention(queries, keys, values, valid_lens, training=False),
                (2, 1, 4))
```

アテンション関数を確認すると、`DotProductAttention` の場合と質的にかなり似た振る舞いが見られる。つまり、選択した有効長 $(2, 6)$ の範囲内の項だけが非ゼロである。

```
d2l.show_heatmaps(d2l.reshape(attention.attention_weights, (1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```

```
d2l.show_heatmaps(d2l.reshape(attention.attention_weights, (1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```

```
d2l.show_heatmaps(d2l.reshape(attention_weights, (1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```

```
d2l.show_heatmaps(d2l.reshape(attention.attention_weights, (1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```

14.3.5 まとめ

この節では、2 つの主要なアテンションのスコアリング関数、すなわち内積アテンションと加法アテンションを導入した。これらは、長さが可変な系列全体を集約するための有効な手段である。特に、内積アテンションは現代の Transformer アーキテクチャの中核をなしている。クエリとキーが異なる長さのベクトルである場合には、代わりに加法アテンションのスコアリング関数を使うことができる。これらの層を最適化することは、近年の進歩の重要な分野の 1 つである。たとえば、NVIDIA の Transformer Library[82] や Megatron (Shoeybi *et al.*, 2019) は、効率的なアテンション機構の変種に大きく依存している。後の節で Transformer を学ぶ際に、これについてさらに詳しく見ていく。

82 

14.3.6 演習

1. `DotProductAttention` のコードを修正して、距離ベースのアテンションを実装せよ。効率的な実装には、キーの二乗ノルム $\|\mathbf{k}_i\|^2$ だけが必要であることに注意せよ。
2. 行列を用いて次元を調整することで、異なる次元のクエリとキーを扱えるように内積アテンションを修正せよ。
3. 計算コストは、キー、クエリ、値の次元およびその個数に対してどのようにスケールするか。メモリ帯域幅の要件についてはどうか。

14.4 バーダナウ注意機構

Section 13.7 で機械翻訳に出会ったとき、私たちは 2 つの RNN に基づく系列変換学習のためのエンコーダ–デコーダアーキテクチャを設計した (Sutskever *et al.*, 2014)。具体的には、RNN エンコーダは可変長の系列を 固定形状のコンテキスト変数へ変換する。その後、RNN デコーダは生成されたトークンとコンテキスト変数に基づいて、出力（ターゲット）系列をトークンごとに生成する。

Fig. 13.7.2 を思い出してほしい。ここではそれを少し詳しくしたものを (Fig. 14.4.1) として再掲する。従来の RNN では、ソース系列に関するすべての関連情報は、エンコーダによって内部の 固定次元の状態表現へと変換される。デコーダは、この状態そのものを、翻訳系列を生成するための完全かつ唯一の情報源として用いる。言い換えると、系列変換機構は中間状態を、入力として与えられたどのような文字列に対しても十分統計量として扱う。

これは短い系列に対してはかなり妥当だが、本や章、あるいは非常に長い文の

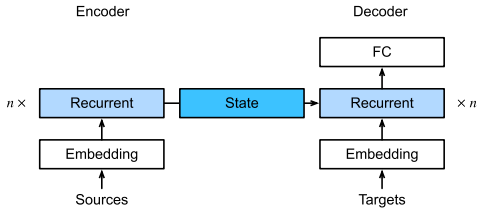


Fig. 14.4.1 系列変換モデル。エンコーダによって生成された状態だけが、エンコーダとデコーダの間で共有される情報である。

ような長い系列に対しては明らかに不可能である。結局のところ、あまり長くはないうちに、中間表現の中にソース系列の重要な情報をすべて格納するための「空間」が足りなくなる。したがって、デコーダは長く複雑な文を翻訳できなくなる。この問題に最初に直面した一人が Graves (2013) であり、彼は手書き文字を生成する RNN を設計しようとした。ソーステキストは任意の長さを取りうるため、彼らは微分可能な注意モデルを設計し、テキスト文字と、はるかに長いペン軌跡とを整列させた。この整列は一方向にのみ進む。これはさらに、音声認識におけるデコードアルゴリズム、たとえば隠れマルコフモデル (Rabiner and Juang, 1993) にも着想を得ている。

整列を学習するというアイデアに触発されて、Bahdanau *et al.* (2014) は、一方向の整列制約 なしの微分可能な注意モデルを提案した。トークンを予測するとき、入力トークンのすべてが関連するとは限らないなら、モデルは現在の予測に関連すると見なされる入力系列の一部にだけ整列（あるいは注意）する。これを用いて、次のトークンを生成する前に現在の状態を更新する。説明だけを見るとさほど目立たないが、この バーダナウ注意機構は、深層学習における過去 10 年で最も影響力のあるアイデアの一つになったと言ってよく、Transformer (Vaswani *et al.*, 2017) や多くの関連する新しいアーキテクチャを生み出した。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import init, np, npx
from mxnet.gluon import rnn, nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from jax import numpy as jnp
import jax
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

14.4.1 モデル

Section 13.7 の系列変換アーキテクチャ、特に (13.7.3) で導入した記法に従う。重要な考え方は、状態、すなわちソース文を要約するコンテキスト変数 $\mathbf{c}$ を固定したままにするのではなく、元のテキスト（エンコーダの隠れ状態 $\mathbf{h}_t$）と、すでに生成されたテキスト（デコーダの隠れ状態 $\mathbf{s}_{t'-1}$）の両方の関数として動的に更新することである。これにより $\mathbf{c}_{t'}$ が得られ、これは任意のデコード時刻 t' の後に更新される。入力系列の長さが T だとしよう。この場合、コンテキスト変数は注意プーリングの出力である：

$$\mathbf{c}_{t'} = \sum_{t=1}^{T} \alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t)\mathbf{h}_t. \tag{14.4.1}$$

ここでは $\mathbf{s}_{t'-1}$ をクエリとして用い、$\mathbf{h}_t$ をキーと値の両方として用いた。$\mathbf{c}_{t'}$ は、その後、状態 $\mathbf{s}_{t'}$ を生成し、新しいトークンを生成するために使われることに注意してほしい。(13.7.3) を参照。特に、注意重み α は (14.3.3) に従って、(14.3.7) で定義された加法的注意スコア関数を用いて計算される。この注意を用いた RNN エンコーダ–デコーダアーキテクチャは Fig. 14.4.2 に示されている。なお、後にこのモデルは、すでに生成されたトークンをデコーダ内でさらなるコンテキストとして含めるように修正された（すなわち、注意の和は T で止まらず、$t'-1$ まで進む）。たとえば、この戦略を音声認識に適用した説明として Chan *et al.* (2015) を参照されたい。

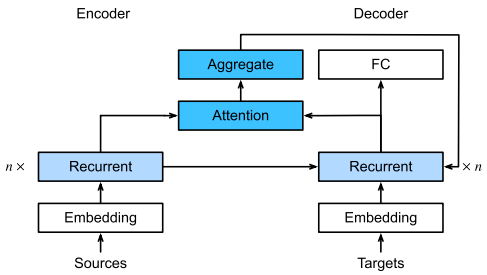


Fig. 14.4.2 バーダナウ注意機構を用いた RNN エンコーダ–デコーダモデルの層。

14.4.2 注意付きデコーダの定義

注意付き RNN エンコーダ–デコーダを実装するには、デコーダを再定義するだけでよい（注意関数から生成済み記号を省くと設計が簡単になる）。まず、`AttentionDecoder` クラスを定義して、注意付きデコーダの基本インターフェースを定めよう。名前からして驚くほどではないが、`AttentionDecoder` クラスである。

```
class AttentionDecoder(d2l.Decoder):  #@save
    """The base attention-based decoder interface."""
    def __init__(self):
        super().__init__()

    @property
    def attention_weights(self):
        raise NotImplementedError
```

```
class AttentionDecoder(d2l.Decoder):  #@save
    """The base attention-based decoder interface."""
    def __init__(self):
        super().__init__()

    @property
    def attention_weights(self):
        raise NotImplementedError
```

```
class Seq2SeqAttentionDecoder(nn.Module):
    vocab_size: int
    embed_size: int
    num_hiddens: int
    num_layers: int
    dropout: float = 0

    def setup(self):
        self.attention = d2l.AdditiveAttention(self.num_hiddens, self.dropout)
        self.embedding = nn.Embed(self.vocab_size, self.embed_size)
        self.dense = nn.Dense(self.vocab_size)
        self.rnn = d2l.GRU(num_hiddens, num_layers, dropout=self.dropout)

    def init_state(self, enc_outputs, enc_valid_lens, *args):
        # Shape of outputs: (num_steps, batch_size, num_hiddens).
        # Shape of hidden_state: (num_layers, batch_size, num_hiddens)
        outputs, hidden_state = enc_outputs
        # Attention Weights are returned as part of state; init with None
        return (outputs.transpose(1, 0, 2), hidden_state, enc_valid_lens)

    @nn.compact
    def __call__(self, X, state, training=False):
        # Shape of enc_outputs: (batch_size, num_steps, num_hiddens).
        # Shape of hidden_state: (num_layers, batch_size, num_hiddens)
        # Ignore Attention value in state
        enc_outputs, hidden_state, enc_valid_lens = state
        # Shape of the output X: (num_steps, batch_size, embed_size)
        X = self.embedding(X).transpose(1, 0, 2)
        outputs, attention_weights = [], []
        for x in X:
            # Shape of query: (batch_size, 1, num_hiddens)
            query = jnp.expand_dims(hidden_state[-1], axis=1)
            # Shape of context: (batch_size, 1, num_hiddens)
            context, attention_w = self.attention(query, enc_outputs,
                                                  enc_outputs, enc_valid_lens,
                                                  training=training)
```

(continues on next page)

(continued from previous page)

```
            # Concatenate on the feature dimension
            x = jnp.concatenate((context, jnp.expand_dims(x, axis=1)), axis=-1)
            # Reshape x as (1, batch_size, embed_size + num_hiddens)
            out, hidden_state = self.rnn(x.transpose(1, 0, 2), hidden_state,
                                         training=training)
            outputs.append(out)
            attention_weights.append(attention_w)

        # Flax sow API is used to capture intermediate variables
        self.sow('intermediates', 'dec_attention_weights', attention_weights)

        # After fully connected layer transformation, shape of outputs:
        # (num_steps, batch_size, vocab_size)
        outputs = self.dense(jnp.concatenate(outputs, axis=0))
        return outputs.transpose(1, 0, 2), [enc_outputs, hidden_state,
                                            enc_valid_lens]
```

```
class AttentionDecoder(d2l.Decoder):  #@save
    """The base attention-based decoder interface."""
    def __init__(self):
        super().__init__()

    @property
    def attention_weights(self):
        raise NotImplementedError
```

Seq2SeqAttentionDecoder クラスで RNN デコーダを実装する必要がある。デコーダの状態は次の要素で初期化される。(i) 注意のキーと値として使われる、すべての時刻におけるエンコーダ最終層の隠れ状態；(ii) デコーダの隠れ状態を初期化するために使われる、最終時刻におけるエンコーダの全層の隠れ状態；(iii) 注意プーリングでパディングトークンを除外するための、エンコーダの有効長。各デコード時刻では、前の時刻で得られたデコーダ最終層の隠れ状態が、注意機構のクエリとして使われる。注意機構の出力と入力埋め込みの両方を連結し、RNN デコーダの入力とする。

```
class Seq2SeqAttentionDecoder(AttentionDecoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.attention = d2l.AdditiveAttention(num_hiddens, dropout)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(
            embed_size + num_hiddens, num_hiddens, num_layers,
            dropout=dropout)
        self.dense = nn.LazyLinear(vocab_size)
        self.apply(d2l.init_seq2seq)

    def init_state(self, enc_outputs, enc_valid_lens):
        # Shape of outputs: (num_steps, batch_size, num_hiddens).
        # Shape of hidden_state: (num_layers, batch_size, num_hiddens)
        outputs, hidden_state = enc_outputs
```

(continues on next page)

(continued from previous page)

```
        return (outputs.permute(1, 0, 2), hidden_state, enc_valid_lens)

    def forward(self, X, state):
        # Shape of enc_outputs: (batch_size, num_steps, num_hiddens).
        # Shape of hidden_state: (num_layers, batch_size, num_hiddens)
        enc_outputs, hidden_state, enc_valid_lens = state
        # Shape of the output X: (num_steps, batch_size, embed_size)
        X = self.embedding(X).permute(1, 0, 2)
        outputs, self._attention_weights = [], []
        for x in X:
            # Shape of query: (batch_size, 1, num_hiddens)
            query = torch.unsqueeze(hidden_state[-1], dim=1)
            # Shape of context: (batch_size, 1, num_hiddens)
            context = self.attention(
                query, enc_outputs, enc_outputs, enc_valid_lens)
            # Concatenate on the feature dimension
            x = torch.cat((context, torch.unsqueeze(x, dim=1)), dim=-1)
            # Reshape x as (1, batch_size, embed_size + num_hiddens)
            out, hidden_state = self.rnn(x.permute(1, 0, 2), hidden_state)
            outputs.append(out)
            self._attention_weights.append(self.attention.attention_weights)
        # After fully connected layer transformation, shape of outputs:
        # (num_steps, batch_size, vocab_size)
        outputs = self.dense(torch.cat(outputs, dim=0))
        return outputs.permute(1, 0, 2), [enc_outputs, hidden_state,
                                          enc_valid_lens]

    @property
    def attention_weights(self):
        return self._attention_weights
```

```
class Seq2SeqAttentionDecoder(AttentionDecoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.attention = d2l.AdditiveAttention(num_hiddens, dropout)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=dropout)
        self.dense = nn.Dense(vocab_size, flatten=False)
        self.initialize(init.Xavier())

    def init_state(self, enc_outputs, enc_valid_lens):
        # Shape of outputs: (num_steps, batch_size, num_hiddens).
        # Shape of hidden_state: (num_layers, batch_size, num_hiddens)
        outputs, hidden_state = enc_outputs
        return (outputs.swapaxes(0, 1), hidden_state, enc_valid_lens)

    def forward(self, X, state):
        # Shape of enc_outputs: (batch_size, num_steps, num_hiddens).
        # Shape of hidden_state: (num_layers, batch_size, num_hiddens)
        enc_outputs, hidden_state, enc_valid_lens = state
        # Shape of the output X: (num_steps, batch_size, embed_size)
        X = self.embedding(X).swapaxes(0, 1)
        outputs, self._attention_weights = [], []
```

(continues on next page)

(continued from previous page)

```
        for x in X:
            # Shape of query: (batch_size, 1, num_hiddens)
            query = np.expand_dims(hidden_state[-1], axis=1)
            # Shape of context: (batch_size, 1, num_hiddens)
            context = self.attention(
                query, enc_outputs, enc_outputs, enc_valid_lens)
            # Concatenate on the feature dimension
            x = np.concatenate((context, np.expand_dims(x, axis=1)), axis=-1)
            # Reshape x as (1, batch_size, embed_size + num_hiddens)
            out, hidden_state = self.rnn(x.swapaxes(0, 1), hidden_state)
            hidden_state = hidden_state[0]
            outputs.append(out)
            self._attention_weights.append(self.attention.attention_weights)
        # After fully connected layer transformation, shape of outputs:
        # (num_steps, batch_size, vocab_size)
        outputs = self.dense(np.concatenate(outputs, axis=0))
        return outputs.swapaxes(0, 1), [enc_outputs, hidden_state,
                                        enc_valid_lens]

    @property
    def attention_weights(self):
        return self._attention_weights
```

```
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
batch_size, num_steps = 4, 7
encoder = d2l.Seq2SeqEncoder(vocab_size, embed_size, num_hiddens, num_layers)
decoder = Seq2SeqAttentionDecoder(vocab_size, embed_size, num_hiddens,
                                  num_layers)
if tab.selected('mxnet'):
    X = d2l.zeros((batch_size, num_steps))
    state = decoder.init_state(encoder(X), None)
    output, state = decoder(X, state)
if tab.selected('pytorch'):
    X = d2l.zeros((batch_size, num_steps), dtype=torch.long)
    state = decoder.init_state(encoder(X), None)
    output, state = decoder(X, state)
if tab.selected('tensorflow'):
    X = tf.zeros((batch_size, num_steps))
    state = decoder.init_state(encoder(X, training=False), None)
    output, state = decoder(X, state, training=False)
if tab.selected('jax'):
    X = jnp.zeros((batch_size, num_steps), dtype=jnp.int32)
    state = decoder.init_state(encoder.init_with_output(d2l.get_key(),
                                                        X, training=False)[0],
                               None)
    (output, state), _ = decoder.init_with_output(d2l.get_key(), X,
                                                  state, training=False)
d2l.check_shape(output, (batch_size, num_steps, vocab_size))
d2l.check_shape(state[0], (batch_size, num_steps, num_hiddens))
d2l.check_shape(state[1][0], (batch_size, num_hiddens))
```

```
class Seq2SeqAttentionDecoder(AttentionDecoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
```

(continues on next page)

(continued from previous page)

```
                 dropout=0):
        super().__init__()
        self.attention = d2l.AdditiveAttention(num_hiddens, num_hiddens,
                                               num_hiddens, dropout)
        self.embedding = tf.keras.layers.Embedding(vocab_size, embed_size)
        self.rnn = tf.keras.layers.RNN(tf.keras.layers.StackedRNNCells(
            [tf.keras.layers.GRUCell(num_hiddens, dropout=dropout)
             for _ in range(num_layers)]), return_sequences=True,
                                       return_state=True)
        self.dense = tf.keras.layers.Dense(vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens):
        # Shape of outputs: (batch_size, num_steps, num_hiddens)
        # Length of list hidden_state is num_layers, where the shape of its
        # element is (batch_size, num_hiddens)
        outputs, hidden_state = enc_outputs
        return (tf.transpose(outputs, (1, 0, 2)), hidden_state,
                enc_valid_lens)

    def call(self, X, state, **kwargs):
        # Shape of output enc_outputs: # (batch_size, num_steps, num_hiddens)
        # Length of list hidden_state is num_layers, where the shape of its
        # element is (batch_size, num_hiddens)
        enc_outputs, hidden_state, enc_valid_lens = state
        # Shape of the output X: (num_steps, batch_size, embed_size)
        X = self.embedding(X)  # Input X has shape: (batch_size, num_steps)
        X = tf.transpose(X, perm=(1, 0, 2))
        outputs, self._attention_weights = [], []
        for x in X:
            # Shape of query: (batch_size, 1, num_hiddens)
            query = tf.expand_dims(hidden_state[-1], axis=1)
            # Shape of context: (batch_size, 1, num_hiddens)
            context = self.attention(query, enc_outputs, enc_outputs,
                                     enc_valid_lens, **kwargs)
            # Concatenate on the feature dimension
            x = tf.concat((context, tf.expand_dims(x, axis=1)), axis=-1)
            out = self.rnn(x, hidden_state, **kwargs)
            hidden_state = out[1:]
            outputs.append(out[0])
            self._attention_weights.append(self.attention.attention_weights)
        # After fully connected layer transformation, shape of outputs:
        # (batch_size, num_steps, vocab_size)
        outputs = self.dense(tf.concat(outputs, axis=1))
        return outputs, [enc_outputs, hidden_state, enc_valid_lens]

    @property
    def attention_weights(self):
        return self._attention_weights
```

以下では、長さ 7 の時系列をそれぞれ持つ 4 つの系列からなるミニバッチを用いて、実装した注意付きデコーダをテストする。

```
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
batch_size, num_steps = 4, 7
```

(continues on next page)

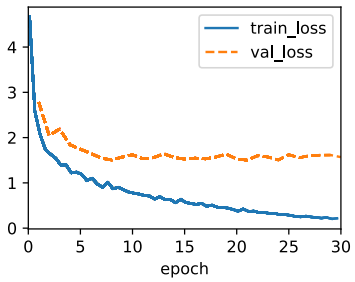
(continued from previous page)

```
encoder = d2l.Seq2SeqEncoder(vocab_size, embed_size, num_hiddens, num_layers)
decoder = Seq2SeqAttentionDecoder(vocab_size, embed_size, num_hiddens,
                                  num_layers)
if tab.selected('mxnet'):
    X = d2l.zeros((batch_size, num_steps))
    state = decoder.init_state(encoder(X), None)
    output, state = decoder(X, state)
if tab.selected('pytorch'):
    X = d2l.zeros((batch_size, num_steps), dtype=torch.long)
    state = decoder.init_state(encoder(X), None)
    output, state = decoder(X, state)
if tab.selected('tensorflow'):
    X = tf.zeros((batch_size, num_steps))
    state = decoder.init_state(encoder(X, training=False), None)
    output, state = decoder(X, state, training=False)
if tab.selected('jax'):
    X = jnp.zeros((batch_size, num_steps), dtype=jnp.int32)
    state = decoder.init_state(encoder.init_with_output(d2l.get_key(),
                                                        X, training=False)[0],
                               None)
    (output, state), _ = decoder.init_with_output(d2l.get_key(), X,
                                                  state, training=False)
d2l.check_shape(output, (batch_size, num_steps, vocab_size))
d2l.check_shape(state[0], (batch_size, num_steps, num_hiddens))
d2l.check_shape(state[1][0], (batch_size, num_hiddens))
```

```
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
batch_size, num_steps = 4, 7
encoder = d2l.Seq2SeqEncoder(vocab_size, embed_size, num_hiddens, num_layers)
decoder = Seq2SeqAttentionDecoder(vocab_size, embed_size, num_hiddens,
                                  num_layers)
if tab.selected('mxnet'):
    X = d2l.zeros((batch_size, num_steps))
    state = decoder.init_state(encoder(X), None)
    output, state = decoder(X, state)
if tab.selected('pytorch'):
    X = d2l.zeros((batch_size, num_steps), dtype=torch.long)
    state = decoder.init_state(encoder(X), None)
    output, state = decoder(X, state)
if tab.selected('tensorflow'):
    X = tf.zeros((batch_size, num_steps))
    state = decoder.init_state(encoder(X, training=False), None)
    output, state = decoder(X, state, training=False)
if tab.selected('jax'):
    X = jnp.zeros((batch_size, num_steps), dtype=jnp.int32)
    state = decoder.init_state(encoder.init_with_output(d2l.get_key(),
                                                        X, training=False)[0],
                               None)
    (output, state), _ = decoder.init_with_output(d2l.get_key(), X,
                                                  state, training=False)
d2l.check_shape(output, (batch_size, num_steps, vocab_size))
d2l.check_shape(state[0], (batch_size, num_steps, num_hiddens))
d2l.check_shape(state[1][0], (batch_size, num_hiddens))
```

```
[08:07:30] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
data = d2l.MTFraEng(batch_size=128)
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2
if tab.selected('mxnet', 'pytorch', 'jax'):
    encoder = d2l.Seq2SeqEncoder(
        len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
    decoder = Seq2SeqAttentionDecoder(
        len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
if tab.selected('mxnet', 'pytorch'):
    model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005)
if tab.selected('jax'):
    model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005, training=True)
if tab.selected('mxnet', 'pytorch', 'jax'):
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        encoder = d2l.Seq2SeqEncoder(
            len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
        decoder = Seq2SeqAttentionDecoder(
            len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
        model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                            lr=0.005)
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1)
trainer.fit(model, data)
```




```
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
batch_size, num_steps = 4, 7
encoder = d2l.Seq2SeqEncoder(vocab_size, embed_size, num_hiddens, num_layers)
decoder = Seq2SeqAttentionDecoder(vocab_size, embed_size, num_hiddens,
                                  num_layers)
if tab.selected('mxnet'):
    X = d2l.zeros((batch_size, num_steps))
    state = decoder.init_state(encoder(X), None)
    output, state = decoder(X, state)
if tab.selected('pytorch'):
    X = d2l.zeros((batch_size, num_steps), dtype=torch.long)
    state = decoder.init_state(encoder(X), None)
```

(continues on next page)

(continued from previous page)

```
    output, state = decoder(X, state)
if tab.selected('tensorflow'):
    X = tf.zeros((batch_size, num_steps))
    state = decoder.init_state(encoder(X, training=False), None)
    output, state = decoder(X, state, training=False)
if tab.selected('jax'):
    X = jnp.zeros((batch_size, num_steps), dtype=jnp.int32)
    state = decoder.init_state(encoder.init_with_output(d2l.get_key(),
                                                        X, training=False)[0],
                               None)
    (output, state), _ = decoder.init_with_output(d2l.get_key(), X,
                                                  state, training=False)
d2l.check_shape(output, (batch_size, num_steps, vocab_size))
d2l.check_shape(state[0], (batch_size, num_steps, num_hiddens))
d2l.check_shape(state[1][0], (batch_size, num_hiddens))
```

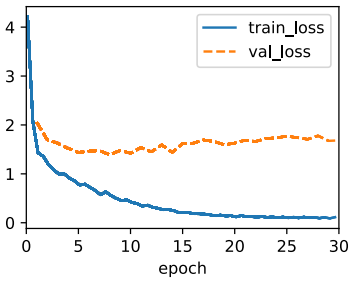
14.4.3 学習

新しいデコーダを定義したので、Section 13.7.6 と同様に進められる。すなわち、ハイパーパラメータを指定し、通常のエンコーダと注意付きデコーダをインスタンス化し、このモデルを機械翻訳のために学習する。

```
data = d2l.MTFraEng(batch_size=128)
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2
if tab.selected('mxnet', 'pytorch', 'jax'):
    encoder = d2l.Seq2SeqEncoder(
        len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
    decoder = Seq2SeqAttentionDecoder(
        len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
if tab.selected('mxnet', 'pytorch'):
    model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005)
if tab.selected('jax'):
    model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005, training=True)
if tab.selected('mxnet', 'pytorch', 'jax'):
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        encoder = d2l.Seq2SeqEncoder(
            len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
        decoder = Seq2SeqAttentionDecoder(
            len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
        model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                            lr=0.005)
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1)
trainer.fit(model, data)
```

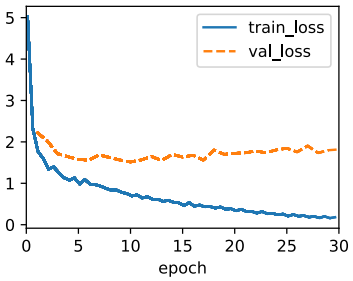
```
data = d2l.MTFraEng(batch_size=128)
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2
if tab.selected('mxnet', 'pytorch', 'jax'):
```

(continues on next page)



(continued from previous page)

```
    encoder = d2l.Seq2SeqEncoder(
        len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
    decoder = Seq2SeqAttentionDecoder(
        len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
if tab.selected('mxnet', 'pytorch'):
    model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005)
if tab.selected('jax'):
    model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005, training=True)
if tab.selected('mxnet', 'pytorch', 'jax'):
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        encoder = d2l.Seq2SeqEncoder(
            len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
        decoder = Seq2SeqAttentionDecoder(
            len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
        model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                            lr=0.005)
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1)
trainer.fit(model, data)
```




```
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    preds, _ = model.predict_step(
        data.build(engs, fras), d2l.try_gpu(), data.num_steps)
```

(continues on next page)

(continued from previous page)

```
if tab.selected('jax'):
    preds, _ = model.predict_step(
        trainer.state.params, data.build(engs, fras), data.num_steps)
for en, fr, p in zip(engs, fras, preds):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
            break
        translation.append(token)
    print(f'{en} => {translation}, bleu,'
          f'{d2l.bleu(" ".join(translation), fr, k=2):.3f}')
```

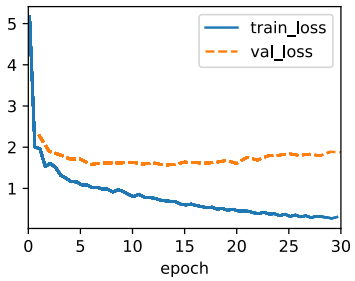
```
go . => ['<unk>', '.'], bleu,0.000
i lost . => ['je', 'me', 'suis', '<unk>', '.'], bleu,0.000
he's calm . => ['assieds-toi', 'ici', '.'], bleu,0.000
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

```
data = d2l.MTFraEng(batch_size=128)
embed_size, num_hiddens, num_layers, dropout = 256, 256, 2, 0.2
if tab.selected('mxnet', 'pytorch', 'jax'):
    encoder = d2l.Seq2SeqEncoder(
        len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
    decoder = Seq2SeqAttentionDecoder(
        len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
if tab.selected('mxnet', 'pytorch'):
    model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005)
if tab.selected('jax'):
    model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.005, training=True)
if tab.selected('mxnet', 'pytorch', 'jax'):
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        encoder = d2l.Seq2SeqEncoder(
            len(data.src_vocab), embed_size, num_hiddens, num_layers, dropout)
        decoder = Seq2SeqAttentionDecoder(
            len(data.tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
        model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                            lr=0.005)
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1)
trainer.fit(model, data)
```

モデルを学習した後、それを用いていくつかの英語文をフランス語に翻訳し、それらの BLEU スコアを計算する。

```
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    preds, _ = model.predict_step(
        data.build(engs, fras), d2l.try_gpu(), data.num_steps)
```

(continues on next page)



(continued from previous page)

```
if tab.selected('jax'):
    preds, _ = model.predict_step(
        trainer.state.params, data.build(engs, fras), data.num_steps)
for en, fr, p in zip(engs, fras, preds):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
            break
        translation.append(token)
    print(f'{en} => {translation}, bleu,'
          f'{d2l.bleu(" ".join(translation), fr, k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
he's calm . => ['je', 'suis', 'calme', '.'], bleu,0.537
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

```
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    preds, _ = model.predict_step(
        data.build(engs, fras), d2l.try_gpu(), data.num_steps)
if tab.selected('jax'):
    preds, _ = model.predict_step(
        trainer.state.params, data.build(engs, fras), data.num_steps)
for en, fr, p in zip(engs, fras, preds):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
            break
        translation.append(token)
    print(f'{en} => {translation}, bleu,'
          f'{d2l.bleu(" ".join(translation), fr, k=2):.3f}')
```

```
go . => ['<unk>', '!'], bleu,0.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
he's calm . => ['tom', 'a', 'gagné', '.'], bleu,0.000
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

```
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    _, dec_attention_weights = model.predict_step(
        data.build([engs[-1]], [fras[-1]]), d2l.try_gpu(), data.num_steps,␣
↪True)
if tab.selected('jax'):
    _, (dec_attention_weights, _) = model.predict_step(
        trainer.state.params, data.build([engs[-1]], [fras[-1]]),
        data.num_steps, True)
attention_weights = d2l.concat(
    [step[0][0][0] for step in dec_attention_weights], 0)
attention_weights = d2l.reshape(attention_weights, (1, 1, -1, data.num_steps))
```

```
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    preds, _ = model.predict_step(
        data.build(engs, fras), d2l.try_gpu(), data.num_steps)
if tab.selected('jax'):
    preds, _ = model.predict_step(
        trainer.state.params, data.build(engs, fras), data.num_steps)
for en, fr, p in zip(engs, fras, preds):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
            break
        translation.append(token)
    print(f'{en} => {translation}, bleu,'
          f'{d2l.bleu(" ".join(translation), fr, k=2):.3f}')
```

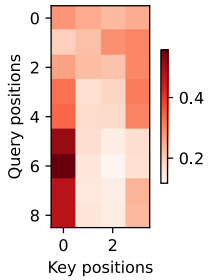
```
go . => ['vas-y', '.'], bleu,0.000
i lost . => ['je', 'me', 'refuse', '.'], bleu,0.000
he's calm . => ['il', 'est', 'mouillé', '.'], bleu,0.658
i'm home . => ['je', 'suis', 'bien', '.'], bleu,0.512
```

最後の英語文を翻訳するときの注意重みを可視化してみよう。各クエリがキー–値のペアに対して非一様な重みを割り当てていることがわかる。これは、各デコードステップで入力系列の異なる部分が注意プーリングで選択的に集約されていることを示している。

```
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    _, dec_attention_weights = model.predict_step(
        data.build([engs[-1]], [fras[-1]]), d2l.try_gpu(), data.num_steps,␣
↪True)
if tab.selected('jax'):
    _, (dec_attention_weights, _) = model.predict_step(
        trainer.state.params, data.build([engs[-1]], [fras[-1]]),
        data.num_steps, True)
attention_weights = d2l.concat(
    [step[0][0][0] for step in dec_attention_weights], 0)
attention_weights = d2l.reshape(attention_weights, (1, 1, -1, data.num_steps))
```

```
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    _, dec_attention_weights = model.predict_step(
        data.build([engs[-1]], [fras[-1]]), d2l.try_gpu(), data.num_steps,␣
↪True)
if tab.selected('jax'):
    _, (dec_attention_weights, _) = model.predict_step(
        trainer.state.params, data.build([engs[-1]], [fras[-1]]),
        data.num_steps, True)
attention_weights = d2l.concat(
    [step[0][0][0] for step in dec_attention_weights], 0)
attention_weights = d2l.reshape(attention_weights, (1, 1, -1, data.num_steps))
```

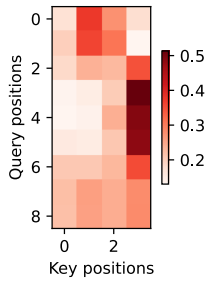
```
# Plus one to include the end-of-sequence token
d2l.show_heatmaps(attention_weights[:, :, :, :len(engs[-1].split()) + 1],
                  xlabel='Key positions', ylabel='Query positions')
```

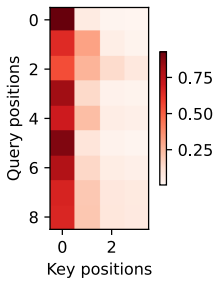



```
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    _, dec_attention_weights = model.predict_step(
        data.build([engs[-1]], [fras[-1]]), d2l.try_gpu(), data.num_steps,␣
↪True)
if tab.selected('jax'):
    _, (dec_attention_weights, _) = model.predict_step(
        trainer.state.params, data.build([engs[-1]], [fras[-1]]),
        data.num_steps, True)
attention_weights = d2l.concat(
    [step[0][0][0] for step in dec_attention_weights], 0)
attention_weights = d2l.reshape(attention_weights, (1, 1, -1, data.num_steps))
```

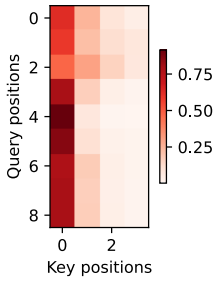
```
# Plus one to include the end-of-sequence token
d2l.show_heatmaps(
    attention_weights[:, :, :, :len(engs[-1].split()) + 1].cpu(),
    xlabel='Key positions', ylabel='Query positions')
```

```
# Plus one to include the end-of-sequence token
d2l.show_heatmaps(
    attention_weights[:, :, :, :len(engs[-1].split()) + 1],
    xlabel='Key positions', ylabel='Query positions')
```






```
# Plus one to include the end-of-sequence token
d2l.show_heatmaps(attention_weights[:, :, :, :len(engs[-1].split()) + 1],
                  xlabel='Key positions', ylabel='Query positions')
```



14.4.4 まとめ

トークンを予測するとき、入力トークンのすべてが関連するとは限らない場合、バーダナウ注意機構を備えた RNN エンコーダ–デコーダは入力系列の異なる部分を選択的に集約する。これは、状態（コンテキスト変数）を加法的注意プーリングの出力として扱うことで実現される。RNN エンコーダ–デコーダでは、バーダナウ注意機構は前時刻のデコーダ隠れ状態をクエリとして扱い、すべての時刻におけるエンコーダ隠れ状態をキーと値の両方として扱う。

14.4.5 演習

1. 実験で GRU を LSTM に置き換えよ。
2. 実験を修正して、加法的注意スコア関数をスケールド・ドット積に置き換えよ。学習効率にどのような影響があるか。

14.5 マルチヘッドアテンション

実際には、同じクエリ、キー、値の集合が与えられたとき、同じアテンション機構の異なる振る舞いから得られる知識をモデルに統合させたい場合がある。たとえば、系列内のさまざまな範囲の依存関係（例：短距離依存と長距離依存）を捉えることである。したがって、アテンション機構が、クエリ、キー、値の異なる表現部分空間を共同で利用できるようにすると有益かもしれない。

この目的のために、単一のアテンションプーリングを行う代わりに、クエリ、キー、値を h 個の独立に学習された線形射影で変換できる。そして、これら h 個の射影されたクエリ、キー、値を並列にアテンションプーリングへ入力する。最後に、h 個のアテンションプーリング出力を連結し、別の学習された線形射影で変換して最終出力を生成する。この設計は マルチヘッドアテンションと呼ばれ、h 個のアテンションプーリング出力のそれぞれを 1 つの ヘッドと呼ぶ (Vaswani *et al.*, 2017)。学習可能な線形変換を行うために全結合層を用いると、Fig. 14.5.1 はマルチヘッドアテンションを説明している。

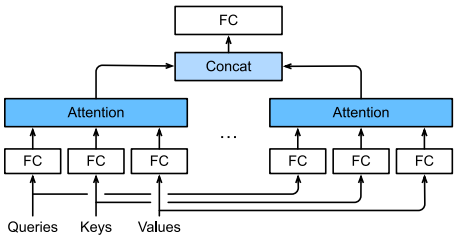


Fig. 14.5.1 マルチヘッドアテンション。複数のヘッドを連結してから線形変換する。

```
from d2l import torch as d2l
import math
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
import math
from mxnet import autograd, np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from jax import numpy as jnp
import jax
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

14.5.1 モデル

マルチヘッドアテンションの実装を示す前に、このモデルを数学的に定式化しよう。クエリ $\mathbf{q} \in \mathbb{R}^{d_q}$、キー $\mathbf{k} \in \mathbb{R}^{d_k}$、値 $\mathbf{v} \in \mathbb{R}^{d_v}$ が与えられたとき、各アテンションヘッド $\mathbf{h}_i$ $(i = 1, \ldots, h)$ は次のように計算される。

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v}, \tag{14.5.1}$$

ここで $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$、$\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$、$\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$ は学習可能なパラメータであり、f はアテンションプーリングである。たとえば、Section 14.3 にある加法アテンションやスケールド・ドット積アテンションが該当する。マルチヘッドアテンションの出力は、h 個のヘッドの連結に対して学習可能なパラメータ $\mathbf{W}_o \in \mathbb{R}^{p_o \times hp_v}$ による別の線形変換である。

$$\mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathbb{R}^{p_o}. \tag{14.5.2}$$

この設計に基づけば、各ヘッドは入力の異なる部分に注意を向けることができる。単純な重み付き平均よりも高度な関数を表現できる。

14.5.2 実装

実装では、マルチヘッドアテンションの各ヘッドにスケールド・ドット積アテンションを選択する。計算コストとパラメータ化コストの大幅な増加を避けるため、$p_q = p_k = p_v = p_o/h$ とする。クエリ、キー、値に対する線形変換の出力数を $p_q h = p_k h = p_v h = p_o$ に設定すれば、h 個のヘッドを並列に計算できることに注意されたい。以下の実装では、p_o は引数 num_hiddens によって指定される。

```
class MultiHeadAttention(d2l.Module):  #@save
    """Multi-head attention."""
    def __init__(self, num_hiddens, num_heads, dropout, bias=False, **kwargs):
        super().__init__()
        self.num_heads = num_heads
```

(continues on next page)

(continued from previous page)

```
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.LazyLinear(num_hiddens, bias=bias)
        self.W_k = nn.LazyLinear(num_hiddens, bias=bias)
        self.W_v = nn.LazyLinear(num_hiddens, bias=bias)
        self.W_o = nn.LazyLinear(num_hiddens, bias=bias)

    def forward(self, queries, keys, values, valid_lens):
        # Shape of queries, keys, or values:
        # (batch_size, no. of queries or key-value pairs, num_hiddens)
        # Shape of valid_lens: (batch_size,) or (batch_size, no. of queries)
        # After transposing, shape of output queries, keys, or values:
        # (batch_size * num_heads, no. of queries or key-value pairs,
        # num_hiddens / num_heads)
        queries = self.transpose_qkv(self.W_q(queries))
        keys = self.transpose_qkv(self.W_k(keys))
        values = self.transpose_qkv(self.W_v(values))

        if valid_lens is not None:
            # On axis 0, copy the first item (scalar or vector) for num_heads
            # times, then copy the next item, and so on
            valid_lens = torch.repeat_interleave(
                valid_lens, repeats=self.num_heads, dim=0)

        # Shape of output: (batch_size * num_heads, no. of queries,
        # num_hiddens / num_heads)
        output = self.attention(queries, keys, values, valid_lens)
        # Shape of output_concat: (batch_size, no. of queries, num_hiddens)
        output_concat = self.transpose_output(output)
        return self.W_o(output_concat)
```

```
class MultiHeadAttention(d2l.Module):  #@save
    """Multi-head attention."""
    def __init__(self, num_hiddens, num_heads, dropout, use_bias=False,
                 **kwargs):
        super().__init__()
        self.num_heads = num_heads
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Dense(num_hiddens, use_bias=use_bias, flatten=False)
        self.W_k = nn.Dense(num_hiddens, use_bias=use_bias, flatten=False)
        self.W_v = nn.Dense(num_hiddens, use_bias=use_bias, flatten=False)
        self.W_o = nn.Dense(num_hiddens, use_bias=use_bias, flatten=False)

    def forward(self, queries, keys, values, valid_lens):
        # Shape of queries, keys, or values:
        # (batch_size, no. of queries or key-value pairs, num_hiddens)
        # Shape of valid_lens: (batch_size,) or (batch_size, no. of queries)
        # After transposing, shape of output queries, keys, or values:
        # (batch_size * num_heads, no. of queries or key-value pairs,
        # num_hiddens / num_heads)
        queries = self.transpose_qkv(self.W_q(queries))
        keys = self.transpose_qkv(self.W_k(keys))
        values = self.transpose_qkv(self.W_v(values))

        if valid_lens is not None:
```

(continues on next page)

(continued from previous page)

```
            # On axis 0, copy the first item (scalar or vector) for num_heads
            # times, then copy the next item, and so on
            valid_lens = valid_lens.repeat(self.num_heads, axis=0)

        # Shape of output: (batch_size * num_heads, no. of queries,
        # num_hiddens / num_heads)
        output = self.attention(queries, keys, values, valid_lens)

        # Shape of output_concat: (batch_size, no. of queries, num_hiddens)
        output_concat = self.transpose_output(output)
        return self.W_o(output_concat)
```

```
class MultiHeadAttention(nn.Module):  #@save
    num_hiddens: int
    num_heads: int
    dropout: float
    bias: bool = False

    def setup(self):
        self.attention = d2l.DotProductAttention(self.dropout)
        self.W_q = nn.Dense(self.num_hiddens, use_bias=self.bias)
        self.W_k = nn.Dense(self.num_hiddens, use_bias=self.bias)
        self.W_v = nn.Dense(self.num_hiddens, use_bias=self.bias)
        self.W_o = nn.Dense(self.num_hiddens, use_bias=self.bias)

    @nn.compact
    def __call__(self, queries, keys, values, valid_lens, training=False):
        # Shape of queries, keys, or values:
        # (batch_size, no. of queries or key-value pairs, num_hiddens)
        # Shape of valid_lens: (batch_size,) or (batch_size, no. of queries)
        # After transposing, shape of output queries, keys, or values:
        # (batch_size * num_heads, no. of queries or key-value pairs,
        # num_hiddens / num_heads)
        queries = self.transpose_qkv(self.W_q(queries))
        keys = self.transpose_qkv(self.W_k(keys))
        values = self.transpose_qkv(self.W_v(values))

        if valid_lens is not None:
            # On axis 0, copy the first item (scalar or vector) for num_heads
            # times, then copy the next item, and so on
            valid_lens = jnp.repeat(valid_lens, self.num_heads, axis=0)

        # Shape of output: (batch_size * num_heads, no. of queries,
        # num_hiddens / num_heads)
        output, attention_weights = self.attention(
            queries, keys, values, valid_lens, training=training)
        # Shape of output_concat: (batch_size, no. of queries, num_hiddens)
        output_concat = self.transpose_output(output)
        return self.W_o(output_concat), attention_weights
```

```
class MultiHeadAttention(d2l.Module):  #@save
    """Multi-head attention."""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
```

(continues on next page)

(continued from previous page)

```
                 num_heads, dropout, bias=False, **kwargs):
        super().__init__()
        self.num_heads = num_heads
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = tf.keras.layers.Dense(num_hiddens, use_bias=bias)
        self.W_k = tf.keras.layers.Dense(num_hiddens, use_bias=bias)
        self.W_v = tf.keras.layers.Dense(num_hiddens, use_bias=bias)
        self.W_o = tf.keras.layers.Dense(num_hiddens, use_bias=bias)

    def call(self, queries, keys, values, valid_lens, **kwargs):
        # Shape of queries, keys, or values:
        # (batch_size, no. of queries or key-value pairs, num_hiddens)
        # Shape of valid_lens: (batch_size,) or (batch_size, no. of queries)
        # After transposing, shape of output queries, keys, or values:
        # (batch_size * num_heads, no. of queries or key-value pairs,
        # num_hiddens / num_heads)
        queries = self.transpose_qkv(self.W_q(queries))
        keys = self.transpose_qkv(self.W_k(keys))
        values = self.transpose_qkv(self.W_v(values))

        if valid_lens is not None:
            # On axis 0, copy the first item (scalar or vector) for num_heads
            # times, then copy the next item, and so on
            valid_lens = tf.repeat(valid_lens, repeats=self.num_heads, axis=0)

        # Shape of output: (batch_size * num_heads, no. of queries,
        # num_hiddens / num_heads)
        output = self.attention(queries, keys, values, valid_lens, **kwargs)

        # Shape of output_concat: (batch_size, no. of queries, num_hiddens)
        output_concat = self.transpose_output(output)
        return self.W_o(output_concat)
```

複数ヘッドの並列計算を可能にするために、上の `MultiHeadAttention` クラスでは以下に定義する 2 つの転置メソッドを使う。具体的には、`transpose_output` メソッドは `transpose_qkv` メソッドの操作を逆にする。

```
@d2l.add_to_class(MultiHeadAttention)  #@save
def transpose_qkv(self, X):
    """Transposition for parallel computation of multiple attention heads."""
    # Shape of input X: (batch_size, no. of queries or key-value pairs,
    # num_hiddens). Shape of output X: (batch_size, no. of queries or
    # key-value pairs, num_heads, num_hiddens / num_heads)
    X = X.reshape(X.shape[0], X.shape[1], self.num_heads, -1)
    # Shape of output X: (batch_size, num_heads, no. of queries or key-value
    # pairs, num_hiddens / num_heads)
    X = X.permute(0, 2, 1, 3)
    # Shape of output: (batch_size * num_heads, no. of queries or key-value
    # pairs, num_hiddens / num_heads)
    return X.reshape(-1, X.shape[2], X.shape[3])

@d2l.add_to_class(MultiHeadAttention)  #@save
def transpose_output(self, X):
```

(continues on next page)

(continued from previous page)

```
    """Reverse the operation of transpose_qkv."""
    X = X.reshape(-1, self.num_heads, X.shape[1], X.shape[2])
    X = X.permute(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)
```

```
@d2l.add_to_class(MultiHeadAttention)  #@save
def transpose_qkv(self, X):
    """Transposition for parallel computation of multiple attention heads."""
    # Shape of input X: (batch_size, no. of queries or key-value pairs,
    # num_hiddens). Shape of output X: (batch_size, no. of queries or
    # key-value pairs, num_heads, num_hiddens / num_heads)
    X = X.reshape(X.shape[0], X.shape[1], self.num_heads, -1)
    # Shape of output X: (batch_size, num_heads, no. of queries or key-value
    # pairs, num_hiddens / num_heads)
    X = X.transpose(0, 2, 1, 3)
    # Shape of output: (batch_size * num_heads, no. of queries or key-value
    # pairs, num_hiddens / num_heads)
    return X.reshape(-1, X.shape[2], X.shape[3])

@d2l.add_to_class(MultiHeadAttention)  #@save
def transpose_output(self, X):
    """Reverse the operation of transpose_qkv."""
    X = X.reshape(-1, self.num_heads, X.shape[1], X.shape[2])
    X = X.transpose(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)
```

```
@d2l.add_to_class(MultiHeadAttention)  #@save
def transpose_qkv(self, X):
    """Transposition for parallel computation of multiple attention heads."""
    # Shape of input X: (batch_size, no. of queries or key-value pairs,
    # num_hiddens). Shape of output X: (batch_size, no. of queries or
    # key-value pairs, num_heads, num_hiddens / num_heads)
    X = X.reshape((X.shape[0], X.shape[1], self.num_heads, -1))
    # Shape of output X: (batch_size, num_heads, no. of queries or key-value
    # pairs, num_hiddens / num_heads)
    X = jnp.transpose(X, (0, 2, 1, 3))
    # Shape of output: (batch_size * num_heads, no. of queries or key-value
    # pairs, num_hiddens / num_heads)
    return X.reshape((-1, X.shape[2], X.shape[3]))

@d2l.add_to_class(MultiHeadAttention)  #@save
def transpose_output(self, X):
    """Reverse the operation of transpose_qkv."""
    X = X.reshape((-1, self.num_heads, X.shape[1], X.shape[2]))
    X = jnp.transpose(X, (0, 2, 1, 3))
    return X.reshape((X.shape[0], X.shape[1], -1))
```

```
@d2l.add_to_class(MultiHeadAttention)  #@save
def transpose_qkv(self, X):
    """Transposition for parallel computation of multiple attention heads."""
    # Shape of input X: (batch_size, no. of queries or key-value pairs,
    # num_hiddens). Shape of output X: (batch_size, no. of queries or
```

(continues on next page)

(continued from previous page)

```
    # key-value pairs, num_heads, num_hiddens / num_heads)
    X = tf.reshape(X, shape=(X.shape[0], X.shape[1], self.num_heads, -1))
    # Shape of output X: (batch_size, num_heads, no. of queries or key-value
    # pairs, num_hiddens / num_heads)
    X = tf.transpose(X, perm=(0, 2, 1, 3))
    # Shape of output: (batch_size * num_heads, no. of queries or key-value
    # pairs, num_hiddens / num_heads)
    return tf.reshape(X, shape=(-1, X.shape[2], X.shape[3]))

@d2l.add_to_class(MultiHeadAttention)  #@save
def transpose_output(self, X):
    """Reverse the operation of transpose_qkv."""
    X = tf.reshape(X, shape=(-1, self.num_heads, X.shape[1], X.shape[2]))
    X = tf.transpose(X, perm=(0, 2, 1, 3))
    return tf.reshape(X, shape=(X.shape[0], X.shape[1], -1))
```

実装した `MultiHeadAttention` クラスを、キーと値が同じであるおもちゃの例を使ってテストしてみよう。その結果、マルチヘッドアテンションの出力形状は (`batch_size`, `num_queries`, `num_hiddens`) になる。

```
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_heads, 0.5)
batch_size, num_queries, num_kvpairs = 2, 4, 6
valid_lens = d2l.tensor([3, 2])
X = d2l.ones((batch_size, num_queries, num_hiddens))
Y = d2l.ones((batch_size, num_kvpairs, num_hiddens))
d2l.check_shape(attention(X, Y, Y, valid_lens),
                (batch_size, num_queries, num_hiddens))
```

```
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_heads, 0.5)
attention.initialize()
```

```
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_heads, 0.5)
```

```
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
                               num_hiddens, num_heads, 0.5)
```

14.5.3 要約

マルチヘッドアテンションは、クエリ、キー、値の異なる表現部分空間を通じて、同じアテンションプーリングの知識を統合する。マルチヘッドアテンションの複数ヘッドを並列に計算するには、適切なテンソル操作が必要である。

14.5.4 演習

1. この実験における複数ヘッドのアテンション重みを可視化せよ。
2. マルチヘッドアテンションに基づく学習済みモデルがあり、予測速度を上げるために重要度の低いアテンションヘッドを剪定したいとする。アテンションヘッドの重要度を測定する実験をどのように設計できるか？

14.6 自己注意機構と位置エンコーディング

深層学習では、系列をエンコードするために CNN や RNN をよく用いる。ここで注意機構を念頭に置くと、トークン列を注意機構に入力し、各ステップで各トークンがそれぞれ独自のクエリ、キー、値を持つと考えることができる。ここで、次の層におけるあるトークンの表現の値を計算するとき、そのトークンは（クエリベクトルを介して）他の任意のトークンに（キーベクトルに基づいて一致を取りながら）注意を向けることができる。クエリとキーの適合度スコアの全体を用いることで、各トークンについて、他のトークンに対する適切な重み付き和を構成し、表現を計算できる。各トークンが互いのトークンに注意を向けるため（デコーダのステップがエンコーダのステップに注意を向ける場合とは異なり）、このようなアーキテクチャは通常 自己注意モデル (Lin *et al.*, 2017, Vaswani *et al.*, 2017) と呼ばれ、別の文脈では *intra-attention* モデル (Cheng *et al.*, 2016, Parikh *et al.*, 2016, Paulus *et al.*, 2017) とも呼ばれる。この節では、系列の順序に関する追加情報も含めた、自己注意を用いる系列エンコーディングについて議論する。

```
from d2l import torch as d2l
import math
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
import math
from mxnet import autograd, np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from jax import numpy as jnp
import jax
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
from d2l import tensorflow as d2l
import numpy as np
import tensorflow as tf
```

14.6.1 自己注意

入力トークン列 $\mathbf{x}_1, \ldots, \mathbf{x}_n$ があり、任意の $\mathbf{x}_i \in \mathbb{R}^d$（$1 \le i \le n$）とする。その自己注意の出力は同じ長さの系列 $\mathbf{y}_1, \ldots, \mathbf{y}_n$ であり、ここで

$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d \tag{14.6.1}$$

は (14.1.1) における注意プーリングの定義に従う。マルチヘッド注意を用いると、次のコード片は形状が（バッチサイズ、時間ステップ数またはトークン列長、d）のテンソルに対する自己注意を計算する。出力テンソルは同じ形状を持つ。

```
num_hiddens, num_heads = 100, 5
attention = d2l.MultiHeadAttention(num_hiddens, num_heads, 0.5)
batch_size, num_queries, valid_lens = 2, 4, d2l.tensor([3, 2])
X = d2l.ones((batch_size, num_queries, num_hiddens))
d2l.check_shape(attention(X, X, X, valid_lens),
                (batch_size, num_queries, num_hiddens))
```

```
num_hiddens, num_heads = 100, 5
attention = d2l.MultiHeadAttention(num_hiddens, num_heads, 0.5)
attention.initialize()
```

```
num_hiddens, num_heads = 100, 5
attention = d2l.MultiHeadAttention(num_hiddens, num_heads, 0.5)
```

```
num_hiddens, num_heads = 100, 5
attention = d2l.MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
                                   num_hiddens, num_heads, 0.5)
```

14.6.2 CNN、RNN、自己注意の比較

n 個のトークンからなる系列を同じ長さの別の系列へ写像するアーキテクチャを比較しよう。ここで各入力トークンまたは出力トークンは d 次元ベクトルで表される。具体的には、CNN、RNN、自己注意を考える。それらの計算量、逐次演算、最大経路長を比較する。逐次演算は並列計算を妨げる一方で、系列位置の任意の組み合わせ間の経路が短いほど、系列内の長距離依存関係を学習しやすくなる (Hochreiter *et al.*, 2001)。

テキスト系列を「一次元画像」とみなしてみよう。同様に、一次元 CNN はテキスト中の n-gram のような局所特徴を処理できる。長さ n の系列を考え、カーネルサイズが k、入力チャネル数と出力チャネル数がともに d の畳み込み層を考え

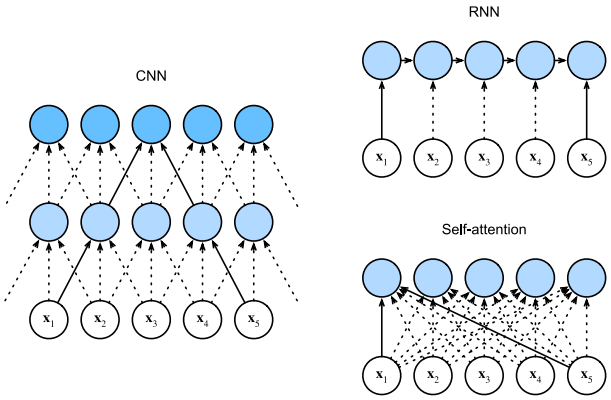


Fig. 14.6.1 CNN（パディングトークンは省略）、RNN、自己注意アーキテクチャの比較。

る。この畳み込み層の計算量は $O(knd^2)$ である。Fig. 14.6.1 が示すように、CNN は階層的であるため、逐次演算は $O(1)$ で済み、最大経路長は $O(n/k)$ である。たとえば、Fig. 14.6.1 では、$\mathbf{x}_1$ と $\mathbf{x}_5$ はカーネルサイズ 3 の 2 層 CNN の受容野内にある。

RNN の隠れ状態を更新するとき、$d \times d$ の重み行列と d 次元の隠れ状態の乗算の計算量は $O(d^2)$ である。系列長が n なので、再帰層の計算量は $O(nd^2)$ である。Fig. 14.6.1 によれば、並列化できない逐次演算が $O(n)$ 回あり、最大経路長も $O(n)$ である。

自己注意では、クエリ、キー、値はすべて $n \times d$ 行列である。(14.3.6) のスケールド・ドット積注意を考えると、$n \times d$ 行列に $d \times n$ 行列を掛け、その後、出力の $n \times n$ 行列に $n \times d$ 行列を掛ける。その結果、自己注意の計算量は $O(n^2 d)$ になる。Fig. 14.6.1 からわかるように、各トークンは自己注意を通じて他の任意のトークンに直接接続されている。したがって、計算は $O(1)$ の逐次演算で並列に行え、最大経路長も $O(1)$ である。

要するに、CNN と自己注意はいずれも並列計算の恩恵を受け、自己注意は最大経路長が最も短い。しかし、系列長に対して二次の計算量を持つため、自己注意は非常に長い系列に対しては極めて遅くなる。

14.6.3 位置エンコーディング

RNN が系列のトークンを 1 つずつ再帰的に処理するのに対し、自己注意は逐次演算を捨てて並列計算を優先する。ただし、自己注意だけでは系列の順序は保持されない。入力系列がどの順序で到着したかをモデルが知っていることが本当に重要な場合、どうすればよいだろうか。

トークンの順序に関する情報を保持するための主流の方法は、各トークンに関連付けられた追加入力としてそれをモデルに表現することである。これらの入力は 位置エンコーディングと呼ばれ、学習可能なものにも、あらかじめ固定さ

れたものにもできる。ここでは、正弦関数と余弦関数に基づく固定位置エンコーディングの簡単な方式を説明する (Vaswani *et al.*, 2017)。

入力表現 $\mathbf{X} \in \mathbb{R}^{n \times d}$ が系列中の n 個のトークンの d 次元埋め込みを含むとする。位置エンコーディングは同じ形状の位置埋め込み行列 $\mathbf{P} \in \mathbb{R}^{n \times d}$ を用いて $\mathbf{X} + \mathbf{P}$ を出力する。その i^{th} 行および $(2j)^{\text{th}}$ または $(2j+1)^{\text{th}}$ 列の要素は

$$\begin{aligned} p_{i,2j} &= \sin\left(\frac{i}{10000^{2j/d}}\right), \\ p_{i,2j+1} &= \cos\left(\frac{i}{10000^{2j/d}}\right). \end{aligned} \tag{14.6.2}$$

一見すると、この三角関数を使った設計は奇妙に見える。この設計の理由を説明する前に、まず次の `PositionalEncoding` クラスで実装してみよう。

```
class PositionalEncoding(nn.Module):  #@save
    """Positional encoding."""
    def __init__(self, num_hiddens, dropout, max_len=1000):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        # Create a long enough P
        self.P = d2l.zeros((1, max_len, num_hiddens))
        X = d2l.arange(max_len, dtype=torch.float32).reshape(
            -1, 1) / torch.pow(10000, torch.arange(
            0, num_hiddens, 2, dtype=torch.float32) / num_hiddens)
        self.P[:, :, 0::2] = torch.sin(X)
        self.P[:, :, 1::2] = torch.cos(X)

    def forward(self, X):
        X = X + self.P[:, :X.shape[1], :].to(X.device)
        return self.dropout(X)
```

```
batch_size, num_queries, valid_lens = 2, 4, d2l.tensor([3, 2])
X = d2l.ones((batch_size, num_queries, num_hiddens))
d2l.check_shape(attention(X, X, X, valid_lens),
                (batch_size, num_queries, num_hiddens))
```

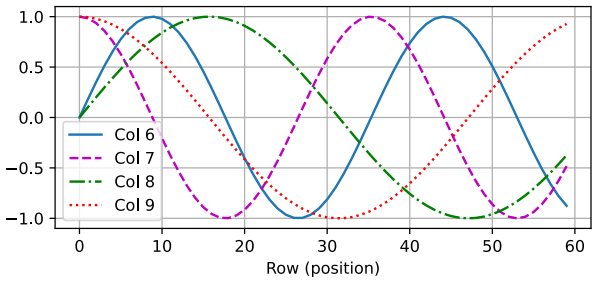
```
[07:15:03] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
batch_size, num_queries, valid_lens = 2, 4, d2l.tensor([3, 2])
X = d2l.ones((batch_size, num_queries, num_hiddens))
d2l.check_shape(attention.init_with_output(d2l.get_key(), X, X, X, valid_lens,
                                           training=False)[0][0],
                (batch_size, num_queries, num_hiddens))
```

```
batch_size, num_queries, valid_lens = 2, 4, tf.constant([3, 2])
X = tf.ones((batch_size, num_queries, num_hiddens))
d2l.check_shape(attention(X, X, X, valid_lens, training=False),
                (batch_size, num_queries, num_hiddens))
```

位置埋め込み行列 $\mathbf{P}$ では、行は系列内の位置に対応し、列は異なる位置エンコーディング次元を表す。以下の例では、位置埋め込み行列の 6^{th} 列と 7^{th} 列は、8^{th} 列と 9^{th} 列よりも高い周波数を持つことがわかる。6^{th} 列と 7^{th} 列（8^{th} 列と 9^{th} 列も同様）の間のオフセットは、正弦関数と余弦関数を交互に用いていることに由来する。

```
encoding_dim, num_steps = 32, 60
pos_encoding = PositionalEncoding(encoding_dim, 0)
X = pos_encoding(d2l.zeros((1, num_steps, encoding_dim)))
P = pos_encoding.P[:, :X.shape[1], :]
d2l.plot(d2l.arange(num_steps), P[0, :, 6:10].T, xlabel='Row (position)',
         figsize=(6, 2.5), legend=["Col %d" % d for d in d2l.arange(6, 10)])
```




```
class PositionalEncoding(nn.Block):  #@save
    """Positional encoding."""
    def __init__(self, num_hiddens, dropout, max_len=1000):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        # Create a long enough P
        self.P = d2l.zeros((1, max_len, num_hiddens))
        X = d2l.arange(max_len).reshape(-1, 1) / np.power(
            10000, np.arange(0, num_hiddens, 2) / num_hiddens)
        self.P[:, :, 0::2] = np.sin(X)
        self.P[:, :, 1::2] = np.cos(X)

    def forward(self, X):
        X = X + self.P[:, :X.shape[1], :].as_in_ctx(X.ctx)
        return self.dropout(X)
```

```
class PositionalEncoding(nn.Module):  #@save
    """Positional encoding."""
    num_hiddens: int
    dropout: float
    max_len: int = 1000

    def setup(self):
        # Create a long enough P
        self.P = d2l.zeros((1, self.max_len, self.num_hiddens))
        X = d2l.arange(self.max_len, dtype=jnp.float32).reshape(
            -1, 1) / jnp.power(10000, jnp.arange(
```

(continues on next page)

(continued from previous page)

```
            0, self.num_hiddens, 2, dtype=jnp.float32) / self.num_hiddens)
        self.P = self.P.at[:, :, 0::2].set(jnp.sin(X))
        self.P = self.P.at[:, :, 1::2].set(jnp.cos(X))

    @nn.compact
    def __call__(self, X, training=False):
        # Flax sow API is used to capture intermediate variables
        self.sow('intermediates', 'P', self.P)
        X = X + self.P[:, :X.shape[1], :]
        return nn.Dropout(self.dropout)(X, deterministic=not training)
```

```
class PositionalEncoding(tf.keras.layers.Layer):  #@save
    """Positional encoding."""
    def __init__(self, num_hiddens, dropout, max_len=1000):
        super().__init__()
        self.dropout = tf.keras.layers.Dropout(dropout)
        # Create a long enough P
        self.P = np.zeros((1, max_len, num_hiddens))
        X = np.arange(max_len, dtype=np.float32).reshape(
            -1,1)/np.power(10000, np.arange(
            0, num_hiddens, 2, dtype=np.float32) / num_hiddens)
        self.P[:, :, 0::2] = np.sin(X)
        self.P[:, :, 1::2] = np.cos(X)

    def call(self, X, **kwargs):
        X = X + self.P[:, :X.shape[1], :]
        return self.dropout(X, **kwargs)
```

絶対的な位置情報

エンコーディング次元に沿って周波数が単調に減少することが絶対的な位置情報とどのように関係するかを見るために、$0, 1, \ldots, 7$ の 2 進表現を出力してみよう。ご覧のように、最下位ビット、下から 2 番目のビット、下から 3 番目のビットは、それぞれ 1 つおき、2 つおき、4 つおきに変化する。

```
for i in range(8):
    print(f'{i} in binary is {i:>03b}')
```

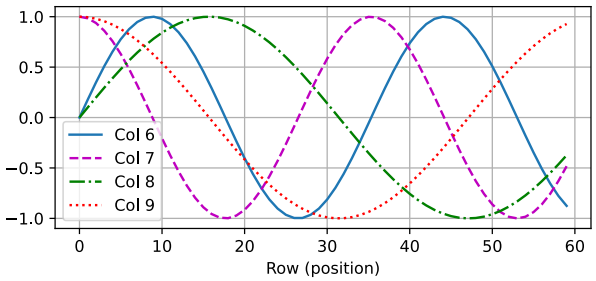
```
0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111
```

```
encoding_dim, num_steps = 32, 60
```

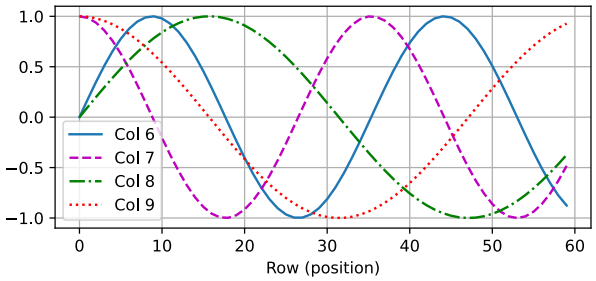
(continues on next page)

(continued from previous page)

```
pos_encoding = PositionalEncoding(encoding_dim, 0)
pos_encoding.initialize()
X = pos_encoding(np.zeros((1, num_steps, encoding_dim)))
P = pos_encoding.P[:, :X.shape[1], :]
d2l.plot(d2l.arange(num_steps), P[0, :, 6:10].T, xlabel='Row (position)',
         figsize=(6, 2.5), legend=["Col %d" % d for d in d2l.arange(6, 10)])
```

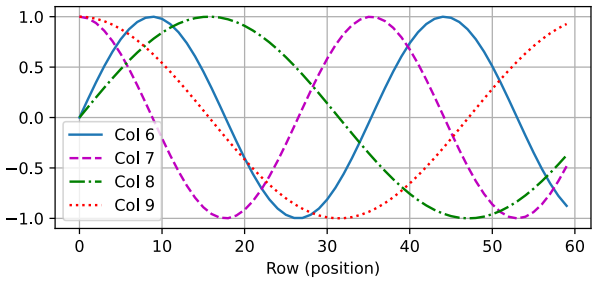



```
encoding_dim, num_steps = 32, 60
pos_encoding = PositionalEncoding(encoding_dim, 0)
params = pos_encoding.init(d2l.get_key(), d2l.zeros((1, num_steps, encoding_
↪dim)))
X, inter_vars = pos_encoding.apply(params, d2l.zeros((1, num_steps, encoding_
↪dim)),
                                   mutable='intermediates')
P = inter_vars['intermediates']['P'][0]  # retrieve intermediate value P
P = P[:, :X.shape[1], :]
d2l.plot(d2l.arange(num_steps), P[0, :, 6:10].T, xlabel='Row (position)',
         figsize=(6, 2.5), legend=["Col %d" % d for d in d2l.arange(6, 10)])
```



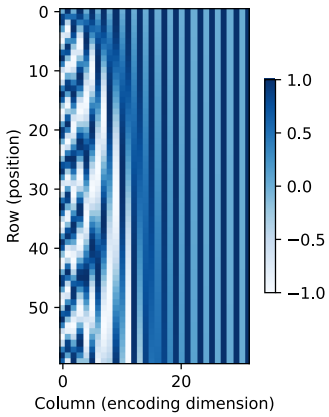

```
encoding_dim, num_steps = 32, 60
pos_encoding = PositionalEncoding(encoding_dim, 0)
X = pos_encoding(tf.zeros((1, num_steps, encoding_dim)), training=False)
P = pos_encoding.P[:, :X.shape[1], :]
d2l.plot(np.arange(num_steps), P[0, :, 6:10].T, xlabel='Row (position)',
         figsize=(6, 2.5), legend=["Col %d" % d for d in np.arange(6, 10)])
```

2 進表現では、上位ビットほど下位ビットよりも周波数が低くなる。同様に、下



のヒートマップが示すように、位置エンコーディングは三角関数を用いてエンコーディング次元に沿って周波数を減少させる。出力は浮動小数点数なので、このような連続表現は 2 進表現よりも空間効率が高い。

```
P = P[0, :, :].unsqueeze(0).unsqueeze(0)
d2l.show_heatmaps(P, xlabel='Column (encoding dimension)',
                  ylabel='Row (position)', figsize=(3.5, 4), cmap='Blues')
```




```
for i in range(8):
    print(f'{i} in binary is {i:>03b}')
```

```
0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111
```

```
for i in range(8):
    print(f'{i} in binary is {i:>03b}')
```

```
0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111
```

```
for i in range(8):
    print(f'{i} in binary is {i:>03b}')
```

```
0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111
```

相対的な位置情報

絶対的な位置情報を捉えることに加えて、上記の位置エンコーディングはモデルが相対位置に基づいて容易に注意を学習することも可能にする。これは、任意の固定位置オフセット δ に対して、位置 $i+\delta$ における位置エンコーディングが位置 i におけるものの線形射影として表せるからである。

この射影は数学的に説明できる。$\omega_j = 1/10000^{2j/d}$ とおくと、(14.6.2) における任意の $(p_{i,2j}, p_{i,2j+1})$ の組は、任意の固定オフセット δ に対して $(p_{i+\delta,2j}, p_{i+\delta,2j+1})$ へ線形射影できる。

$$\begin{aligned}\begin{bmatrix} \cos(\delta\omega_j) & \sin(\delta\omega_j) \\ -\sin(\delta\omega_j) & \cos(\delta\omega_j) \end{bmatrix} \begin{bmatrix} p_{i,2j} \\ p_{i,2j+1} \end{bmatrix} &= \begin{bmatrix} \cos(\delta\omega_j)\sin(i\omega_j) + \sin(\delta\omega_j)\cos(i\omega_j) \\ -\sin(\delta\omega_j)\sin(i\omega_j) + \cos(\delta\omega_j)\cos(i\omega_j) \end{bmatrix} \\ &= \begin{bmatrix} \sin\left((i+\delta)\omega_j\right) \\ \cos\left((i+\delta)\omega_j\right) \end{bmatrix} \\ &= \begin{bmatrix} p_{i+\delta,2j} \\ p_{i+\delta,2j+1} \end{bmatrix},\end{aligned} \tag{14.6.3}$$

ここで、2×2 の射影行列はどの位置インデックス i にも依存しない。

14.6.4 まとめ

自己注意では、クエリ、キー、値はすべて同じ場所から来る。CNN と自己注意はいずれも並列計算の恩恵を受け、自己注意は最大経路長が最も短い。しかし、系列長に対して二次の計算量を持つため、自己注意は非常に長い系列

に対しては極めて遅くなる。系列順序の情報を使うには、入力表現に位置エンコーディングを加えることで、絶対的または相対的な位置情報を注入できる。

14.6.5 演習

1. 位置エンコーディングを用いた自己注意層を積み重ねることで系列を表現する深いアーキテクチャを設計するとする。どのような問題が起こりうるだろうか。
2. 学習可能な位置エンコーディング手法を設計できるか？
3. 自己注意で比較されるクエリとキーの間の異なるオフセットに応じて、異なる学習済み埋め込みを割り当てることはできるか？ヒント：相対位置埋め込みを参照されたい (Huang *et al.*, 2018, Shaw *et al.*, 2018)。

14.7 Transformer アーキテクチャ

Section 14.6.2 では、CNN、RNN、自己注意を比較した。特に、自己注意は並列計算と最短の最大経路長の両方を備えている。したがって、自己注意を用いて深いアーキテクチャを設計するのは魅力的である。入力表現に対してなお RNN に依存していた以前の自己注意モデル (Cheng *et al.*, 2016, Lin *et al.*, 2017, Paulus *et al.*, 2017) とは異なり、Transformer モデルは畳み込み層も再帰層も使わず、注意機構のみに基づいている (Vaswani *et al.*, 2017)。もともとはテキストデータに対する系列変換学習のために提案されたが、Transformer は言語、視覚、音声、強化学習など、現代の深層学習の幅広い応用分野で広く使われるようになっている。

```
from d2l import torch as d2l
import math
import pandas as pd
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
import math
from mxnet import autograd, init, np, npx
from mxnet.gluon import nn
import pandas as pd
npx.set_np()
```

```
from d2l import jax as d2l
from flax import linen as nn
from jax import numpy as jnp
```

(continues on next page)

(continued from previous page)

```
import jax
import math
import pandas as pd
```

```
from d2l import tensorflow as d2l
import numpy as np
import pandas as pd
import tensorflow as tf
```

14.7.1 モデル

エンコーダ–デコーダアーキテクチャの一例として、Transformer の全体アーキテクチャを Fig. 14.7.1 に示す。見てわかるように、Transformer はエンコーダとデコーダから構成される。Fig. 14.4.2 の Bahdanau 注意による系列変換学習とは対照的に、入力（source）系列と出力（target）系列の埋め込みには、自己注意に基づくモジュールを積み重ねたエンコーダとデコーダに入力する前に、位置エンコーディングが加えられる。

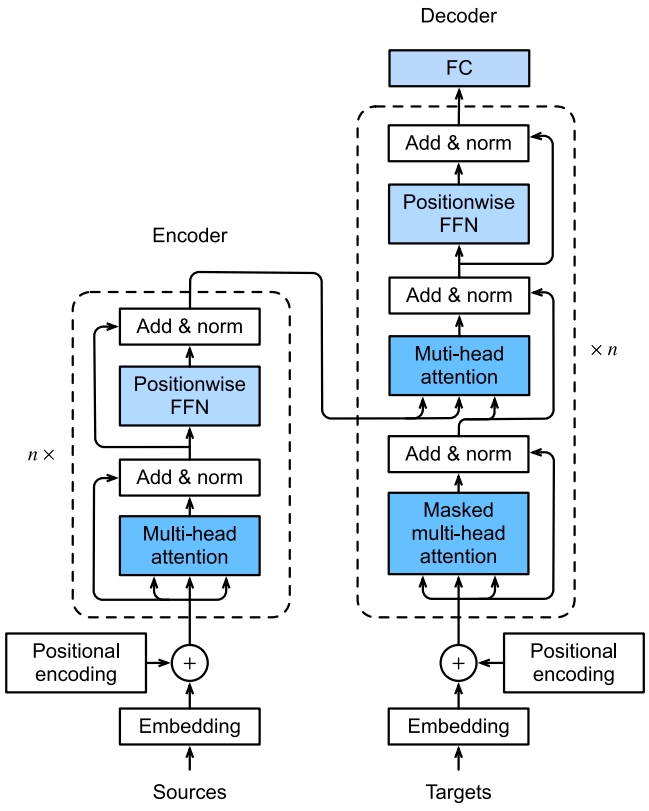


Fig. 14.7.1 The Transformer architecture.

ここで、Fig. 14.7.1 における Transformer アーキテクチャの概要を説明する。高レベルでは、Transformer エンコーダは複数の同一層のスタックであり、各層は 2 つのサブレイヤー（いずれも sublayer と表記）を持つ。1 つ目はマルチヘッド自己注意プーリングであり、2 つ目は位置ごとのフィードフォワードネットワークである。具体的には、エンコーダの自己注意では、クエリ、キー、値はすべて前のエンコーダ層の出力から得られる。Section 11.6 の ResNet 設計に着想を得て、両方のサブレイヤーの周囲に残差接続が用いられる。Transformer では、系列の任意の位置にある任意の入力 $\mathbf{x} \in \mathbb{R}^d$ に対して、残差接続 $\mathbf{x}+\text{sublayer}(\mathbf{x}) \in \mathbb{R}^d$ が可能であるように、$\text{sublayer}(\mathbf{x}) \in \mathbb{R}^d$ を要求する。この残差接続による加算の直後に層正規化が続く (Ba *et al.*, 2016)。その結果、Transformer エンコーダは入力系列の各位置に対して d 次元のベクトル表現を出力する。

Transformer デコーダもまた、残差接続と層正規化を備えた複数の同一層のスタックである。エンコーダで説明した 2 つのサブレイヤーに加えて、デコーダはその間にエンコーダ–デコーダ注意と呼ばれる 3 つ目のサブレイヤーを挿入する。エンコーダ–デコーダ注意では、クエリはデコーダの自己注意サブレイヤーの出力から得られ、キーと値は Transformer エンコーダの出力から得られる。デコーダの自己注意では、クエリ、キー、値はすべて前のデコーダ層の出力から得られる。ただし、デコーダの各位置はその位置までのデコーダ内のすべての位置にのみ注意を向けることが許される。このマスク付き注意は自己回帰性を保ち、予測が生成済みの出力トークンのみに依存することを保証する。

すでに Section 14.5 でスケールド・ドット積に基づくマルチヘッド注意と、Section 14.6.3 で位置エンコーディングを説明し実装した。以下では、Transformer モデルの残りの部分を実装する。

14.7.2 位置ごとのフィードフォワードネットワーク

位置ごとのフィードフォワードネットワークは、同じ MLP を用いてすべての系列位置の表現を変換する。このため、これを位置ごとと呼ぶ。以下の実装では、形状が（バッチサイズ、時間ステップ数またはトークン単位の系列長、隠れユニット数または特徴次元）である入力 `X` は、2 層 MLP によって形状が（バッチサイズ、時間ステップ数、`ffn_num_outputs`）の出力テンソルへ変換される。

```
class PositionWiseFFN(nn.Module):  #@save
    """The positionwise feed-forward network."""
    def __init__(self, ffn_num_hiddens, ffn_num_outputs):
        super().__init__()
        self.dense1 = nn.LazyLinear(ffn_num_hiddens)
        self.relu = nn.ReLU()
        self.dense2 = nn.LazyLinear(ffn_num_outputs)

    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

```
class PositionWiseFFN(nn.Block):  #@save
```

(continues on next page)

(continued from previous page)

```
    """The positionwise feed-forward network."""
    def __init__(self, ffn_num_hiddens, ffn_num_outputs):
        super().__init__()
        self.dense1 = nn.Dense(ffn_num_hiddens, flatten=False,
                               activation='relu')
        self.dense2 = nn.Dense(ffn_num_outputs, flatten=False)

    def forward(self, X):
        return self.dense2(self.dense1(X))
```

```
class PositionWiseFFN(nn.Module):  #@save
    """The positionwise feed-forward network."""
    ffn_num_hiddens: int
    ffn_num_outputs: int

    def setup(self):
        self.dense1 = nn.Dense(self.ffn_num_hiddens)
        self.dense2 = nn.Dense(self.ffn_num_outputs)

    def __call__(self, X):
        return self.dense2(nn.relu(self.dense1(X)))
```

```
class PositionWiseFFN(tf.keras.layers.Layer):  #@save
    """The positionwise feed-forward network."""
    def __init__(self, ffn_num_hiddens, ffn_num_outputs):
        super().__init__()
        self.dense1 = tf.keras.layers.Dense(ffn_num_hiddens)
        self.relu = tf.keras.layers.ReLU()
        self.dense2 = tf.keras.layers.Dense(ffn_num_outputs)

    def call(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

次の例は、テンソルの最内側の次元が変化することを示している。その変化先は位置ごとのフィードフォワードネットワークの出力数である。同じ MLP がすべての位置で変換を行うため、それらすべての位置で入力が同じなら、出力も同一になる。

```
ffn = PositionWiseFFN(4, 8)
ffn.eval()
ffn(d2l.ones((2, 3, 4)))[0]
```

```
tensor([[-0.4807,  0.1850,  0.1983,  0.0662,  0.3201,  0.5081,  0.1139, -0.
↪1333],
        [-0.4807,  0.1850,  0.1983,  0.0662,  0.3201,  0.5081,  0.1139, -0.
↪1333],
        [-0.4807,  0.1850,  0.1983,  0.0662,  0.3201,  0.5081,  0.1139, -0.
↪1333]],
       grad_fn=<SelectBackward0>)
```

```
ffn = PositionWiseFFN(4, 8)
ffn.initialize()
ffn(np.ones((2, 3, 4)))[0]
```

```
[07:43:27] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
array([[ 0.00239431,  0.00927085, -0.00021069, -0.00923989, -0.0082903 ,
        -0.00162741,  0.00659031,  0.00023905],
       [ 0.00239431,  0.00927085, -0.00021069, -0.00923989, -0.0082903 ,
        -0.00162741,  0.00659031,  0.00023905],
       [ 0.00239431,  0.00927085, -0.00021069, -0.00923989, -0.0082903 ,
        -0.00162741,  0.00659031,  0.00023905]])
```

```
ffn = PositionWiseFFN(4, 8)
ffn.init_with_output(d2l.get_key(), jnp.ones((2, 3, 4)))[0][0]
```

```
Array([[-0.5607809 ,  0.00511155, -0.15698639, -0.46223652, -0.06088082,
         0.41905117, -0.38441786,  0.28503913],
       [-0.5607809 ,  0.00511155, -0.15698639, -0.46223652, -0.06088082,
         0.41905117, -0.38441786,  0.28503913],
       [-0.5607809 ,  0.00511155, -0.15698639, -0.46223652, -0.06088082,
         0.41905117, -0.38441786,  0.28503913]], dtype=float32)
```

```
ffn = PositionWiseFFN(4, 8)
ffn(tf.ones((2, 3, 4)))[0]
```

```
<tf.Tensor: shape=(3, 8), dtype=float32, numpy=
array([[-0.994034  ,  0.57995766, -0.07282299, -0.5814867 , -0.43248075,
        -0.7298058 , -0.35487527, -0.3816934 ],
       [-0.994034  ,  0.57995766, -0.07282299, -0.5814867 , -0.43248075,
        -0.7298058 , -0.35487527, -0.3816934 ],
       [-0.994034  ,  0.57995766, -0.07282299, -0.5814867 , -0.43248075,
        -0.7298058 , -0.35487527, -0.3816934 ]], dtype=float32)>
```

14.7.3 残差接続と層正規化

ここでは、Fig. 14.7.1 の「add & norm」コンポーネントに注目しよう。この節の冒頭で述べたように、これは残差接続の直後に層正規化が続く構成である。どちらも効果的な深層アーキテクチャにとって重要である。

Section 11.5 では、バッチ正規化がミニバッチ内の各例にわたってどのように再中心化と再スケーリングを行うかを説明した。Section 11.5.2 で議論したように、層正規化はバッチ正規化と同じだが、前者は特徴次元に沿って正規化する点が異なる。そのため、スケール不変性とバッチサイズ不変性の利点を持つ。コンピュータビジョンで広く使われているにもかかわらず、バッチ正規化は、

入力がしばしば可変長系列である自然言語処理タスクでは、経験的に層正規化ほど有効でないことが多い。

次のコード片は、層正規化とバッチ正規化による異なる次元に沿った正規化を比較する。

```
ln = nn.LayerNorm(2)
bn = nn.LazyBatchNorm1d()
X = d2l.tensor([[1, 2], [2, 3]], dtype=torch.float32)
# Compute mean and variance from X in the training mode
print('layer norm:', ln(X), '\nbatch norm:', bn(X))
```

```
layer norm: tensor([[-1.0000,  1.0000],
        [-1.0000,  1.0000]], grad_fn=<NativeLayerNormBackward0>)
batch norm: tensor([[-1.0000, -1.0000],
        [ 1.0000,  1.0000]], grad_fn=<NativeBatchNormBackward0>)
```

```
ln = nn.LayerNorm()
ln.initialize()
bn = nn.BatchNorm()
bn.initialize()
X = d2l.tensor([[1, 2], [2, 3]])
# Compute mean and variance from X in the training mode
with autograd.record():
    print('layer norm:', ln(X), '\nbatch norm:', bn(X))
```

```
layer norm: [[-0.99998  0.99998]
 [-0.99998  0.99998]]
batch norm: [[-0.99998 -0.99998]
 [ 0.99998  0.99998]]
```

```
ln = nn.LayerNorm()
bn = nn.BatchNorm()
X = d2l.tensor([[1, 2], [2, 3]], dtype=d2l.float32)
# Compute mean and variance from X in the training mode
print('layer norm:', ln.init_with_output(d2l.get_key(), X)[0],
      '\nbatch norm:', bn.init_with_output(d2l.get_key(), X,
                                           use_running_average=False)[0])
```

```
layer norm: [[-0.9999979  0.9999979]
 [-0.9999979  0.9999979]]
batch norm: [[-0.9999799 -0.9999799]
 [ 0.9999799  0.9999799]]
```

```
ln = tf.keras.layers.LayerNormalization()
bn = tf.keras.layers.BatchNormalization()
X = tf.constant([[1, 2], [2, 3]], dtype=tf.float32)
print('layer norm:', ln(X), '\nbatch norm:', bn(X, training=True))
```

```
layer norm: tf.Tensor(
[[-0.998006   0.9980061]
 [-0.9980061  0.998006 ]], shape=(2, 2), dtype=float32)
batch norm: tf.Tensor(
[[-0.998006   -0.9980061 ]
 [ 0.9980061   0.99800587]], shape=(2, 2), dtype=float32)
```

ここで、残差接続の後に層正規化を行う AddNorm クラスを実装できる。正則化のためにドロップアウトも適用する。

```
class AddNorm(nn.Module):  #@save
    """The residual connection followed by layer normalization."""
    def __init__(self, norm_shape, dropout):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(norm_shape)

    def forward(self, X, Y):
        return self.ln(self.dropout(Y) + X)
```

```
class AddNorm(nn.Block):  #@save
    """The residual connection followed by layer normalization."""
    def __init__(self, dropout):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm()

    def forward(self, X, Y):
        return self.ln(self.dropout(Y) + X)
```

```
class AddNorm(nn.Module):  #@save
    """The residual connection followed by layer normalization."""
    dropout: int

    @nn.compact
    def __call__(self, X, Y, training=False):
        return nn.LayerNorm()(
            nn.Dropout(self.dropout)(Y, deterministic=not training) + X)
```

```
class AddNorm(tf.keras.layers.Layer):  #@save
    """The residual connection followed by layer normalization."""
    def __init__(self, norm_shape, dropout):
        super().__init__()
        self.dropout = tf.keras.layers.Dropout(dropout)
        self.ln = tf.keras.layers.LayerNormalization(norm_shape)

    def call(self, X, Y, **kwargs):
        return self.ln(self.dropout(Y, **kwargs) + X)
```

残差接続では、加算演算の後も出力テンソルが同じ形状になるように、2 つの入

力が同じ形状であることが必要である。そのため、出力テンソルも加算後に同じ形状を持つ。

```
add_norm = AddNorm(4, 0.5)
shape = (2, 3, 4)
d2l.check_shape(add_norm(d2l.ones(shape), d2l.ones(shape)), shape)
```

```
add_norm = AddNorm(0.5)
add_norm.initialize()
shape = (2, 3, 4)
d2l.check_shape(add_norm(d2l.ones(shape), d2l.ones(shape)), shape)
```

```
add_norm = AddNorm(0.5)
shape = (2, 3, 4)
output, _ = add_norm.init_with_output(d2l.get_key(), d2l.ones(shape),
                                      d2l.ones(shape))
d2l.check_shape(output, shape)
```

```
# Normalized_shape is: [i for i in range(len(input.shape))][1:]
add_norm = AddNorm([1, 2], 0.5)
shape = (2, 3, 4)
d2l.check_shape(add_norm(tf.ones(shape), tf.ones(shape), training=False),
                shape)
```

14.7.4 エンコーダ

Transformer エンコーダを構成するための必要な要素がすべてそろったので、まずはエンコーダ内の 1 層を実装しよう。以下の `TransformerEncoderBlock` クラスは 2 つのサブレイヤーを含む。すなわち、マルチヘッド自己注意と位置ごとのフィードフォワードネットワークであり、両方のサブレイヤーの周囲に残差接続と層正規化が用いられる。

```
class TransformerEncoderBlock(nn.Module):  #@save
    """The Transformer encoder block."""
    def __init__(self, num_hiddens, ffn_num_hiddens, num_heads, dropout,
                 use_bias=False):
        super().__init__()
        self.attention = d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                dropout, use_bias)
        self.addnorm1 = AddNorm(num_hiddens, dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(num_hiddens, dropout)

    def forward(self, X, valid_lens):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
        return self.addnorm2(Y, self.ffn(Y))
```

```
class TransformerEncoderBlock(nn.Block):  #@save
```

(continues on next page)

(continued from previous page)

```
    """The Transformer encoder block."""
    def __init__(self, num_hiddens, ffn_num_hiddens, num_heads, dropout,
                 use_bias=False):
        super().__init__()
        self.attention = d2l.MultiHeadAttention(
            num_hiddens, num_heads, dropout, use_bias)
        self.addnorm1 = AddNorm(dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(dropout)

    def forward(self, X, valid_lens):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
        return self.addnorm2(Y, self.ffn(Y))
```

```
class TransformerEncoderBlock(nn.Module):  #@save
    """The Transformer encoder block."""
    num_hiddens: int
    ffn_num_hiddens: int
    num_heads: int
    dropout: float
    use_bias: bool = False

    def setup(self):
        self.attention = d2l.MultiHeadAttention(self.num_hiddens, self.num_
↪heads,
                                                self.dropout, self.use_bias)
        self.addnorm1 = AddNorm(self.dropout)
        self.ffn = PositionWiseFFN(self.ffn_num_hiddens, self.num_hiddens)
        self.addnorm2 = AddNorm(self.dropout)

    def __call__(self, X, valid_lens, training=False):
        output, attention_weights = self.attention(X, X, X, valid_lens,
                                                   training=training)
        Y = self.addnorm1(X, output, training=training)
        return self.addnorm2(Y, self.ffn(Y), training=training), attention_
↪weights
```

```
class TransformerEncoderBlock(tf.keras.layers.Layer):  #@save
    """The Transformer encoder block."""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_hiddens, num_heads, dropout, bias=False):
        super().__init__()
        self.attention = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout,
            bias)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(norm_shape, dropout)

    def call(self, X, valid_lens, **kwargs):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens, **kwargs),
                          **kwargs)
        return self.addnorm2(Y, self.ffn(Y), **kwargs)
```

見てわかるように、Transformer エンコーダのどの層も入力の形状を変えない。

```
X = d2l.ones((2, 100, 24))
valid_lens = d2l.tensor([3, 2])
encoder_blk = TransformerEncoderBlock(24, 48, 8, 0.5)
encoder_blk.eval()
d2l.check_shape(encoder_blk(X, valid_lens), X.shape)
```

```
X = d2l.ones((2, 100, 24))
valid_lens = d2l.tensor([3, 2])
encoder_blk = TransformerEncoderBlock(24, 48, 8, 0.5)
encoder_blk.initialize()
d2l.check_shape(encoder_blk(X, valid_lens), X.shape)
```

```
X = jnp.ones((2, 100, 24))
valid_lens = jnp.array([3, 2])
encoder_blk = TransformerEncoderBlock(24, 48, 8, 0.5)
(output, _), _ = encoder_blk.init_with_output(d2l.get_key(), X, valid_lens,
                                              training=False)
d2l.check_shape(output, X.shape)
```

```
X = tf.ones((2, 100, 24))
valid_lens = tf.constant([3, 2])
norm_shape = [i for i in range(len(X.shape))][1:]
encoder_blk = TransformerEncoderBlock(24, 24, 24, 24, norm_shape, 48, 8, 0.5)
d2l.check_shape(encoder_blk(X, valid_lens, training=False), X.shape)
```

以下の Transformer エンコーダの実装では、上記の `TransformerEncoderBlock` クラスの `num_blks` 個のインスタンスを積み重ねる。固定位置エンコーディングの値は常に -1 と 1 の間にあるため、入力埋め込みと位置エンコーディングを加算する前に、学習可能な入力埋め込みの値に埋め込み次元の平方根を掛けて再スケーリングする。

```
class TransformerEncoder(d2l.Encoder):  #@save
    """The Transformer encoder."""
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens,
                 num_heads, num_blks, dropout, use_bias=False):
        super().__init__()
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_blks):
            self.blks.add_module("block"+str(i), TransformerEncoderBlock(
                num_hiddens, ffn_num_hiddens, num_heads, dropout, use_bias))

    def forward(self, X, valid_lens):
        # Since positional encoding values are between -1 and 1, the embedding
        # values are multiplied by the square root of the embedding dimension
```

(continues on next page)

(continued from previous page)

```
        # to rescale before they are summed up
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[
                i] = blk.attention.attention.attention_weights
        return X
```

```
class TransformerEncoder(d2l.Encoder):  #@save
    """The Transformer encoder."""
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens,
                 num_heads, num_blks, dropout, use_bias=False):
        super().__init__()
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for _ in range(num_blks):
            self.blks.add(TransformerEncoderBlock(
                num_hiddens, ffn_num_hiddens, num_heads, dropout, use_bias))
        self.initialize()

    def forward(self, X, valid_lens):
        # Since positional encoding values are between -1 and 1, the embedding
        # values are multiplied by the square root of the embedding dimension
        # to rescale before they are summed up
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[
                i] = blk.attention.attention.attention_weights
        return X
```

```
class TransformerEncoder(d2l.Encoder):  #@save
    """The Transformer encoder."""
    vocab_size: int
    num_hiddens:int
    ffn_num_hiddens: int
    num_heads: int
    num_blks: int
    dropout: float
    use_bias: bool = False

    def setup(self):
        self.embedding = nn.Embed(self.vocab_size, self.num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(self.num_hiddens, self.
↪dropout)
        self.blks = [TransformerEncoderBlock(self.num_hiddens,
                                             self.ffn_num_hiddens,
                                             self.num_heads,
                                             self.dropout, self.use_bias)
```

(continues on next page)

(continued from previous page)

```
                    for _ in range(self.num_blks)]

    def __call__(self, X, valid_lens, training=False):
        # Since positional encoding values are between -1 and 1, the embedding
        # values are multiplied by the square root of the embedding dimension
        # to rescale before they are summed up
        X = self.embedding(X) * math.sqrt(self.num_hiddens)
        X = self.pos_encoding(X, training=training)
        attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X, attention_w = blk(X, valid_lens, training=training)
            attention_weights[i] = attention_w
        # Flax sow API is used to capture intermediate variables
        self.sow('intermediates', 'enc_attention_weights', attention_weights)
        return X
```

```
class TransformerEncoder(d2l.Encoder):  #@save
    """The Transformer encoder."""
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_hiddens, num_heads,
                 num_blks, dropout, bias=False):
        super().__init__()
        self.num_hiddens = num_hiddens
        self.embedding = tf.keras.layers.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = [TransformerEncoderBlock(
            key_size, query_size, value_size, num_hiddens, norm_shape,
            ffn_num_hiddens, num_heads, dropout, bias) for _ in range(
            num_blks)]

    def call(self, X, valid_lens, **kwargs):
        # Since positional encoding values are between -1 and 1, the embedding
        # values are multiplied by the square root of the embedding dimension
        # to rescale before they are summed up
        X = self.pos_encoding(self.embedding(X) * tf.math.sqrt(
            tf.cast(self.num_hiddens, dtype=tf.float32)), **kwargs)
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens, **kwargs)
            self.attention_weights[
                i] = blk.attention.attention.attention_weights
        return X
```

以下では、ハイパーパラメータを指定して 2 層の Transformer エンコーダを作成する。Transformer エンコーダの出力形状は（バッチサイズ、時間ステップ数、`num_hiddens`）である。

```
encoder = TransformerEncoder(200, 24, 48, 8, 2, 0.5)
d2l.check_shape(encoder(d2l.ones((2, 100), dtype=torch.long), valid_lens),
                (2, 100, 24))
```

```
encoder = TransformerEncoder(200, 24, 48, 8, 2, 0.5)
d2l.check_shape(encoder(np.ones((2, 100)), valid_lens), (2, 100, 24))
```

```
encoder = TransformerEncoder(200, 24, 48, 8, 2, 0.5)
d2l.check_shape(encoder.init_with_output(d2l.get_key(),
                                         jnp.ones((2, 100), dtype=jnp.int32),
                                         valid_lens)[0],
                (2, 100, 24))
```

```
encoder = TransformerEncoder(200, 24, 24, 24, 24, [1, 2], 48, 8, 2, 0.5)
d2l.check_shape(encoder(tf.ones((2, 100)), valid_lens, training=False),
                (2, 100, 24))
```

14.7.5 デコーダ

Fig. 14.7.1 に示すように、Transformer デコーダは複数の同一層から構成される。各層は以下の `TransformerDecoderBlock` クラスで実装され、3 つのサブレイヤーを含む。すなわち、デコーダ自己注意、エンコーダ–デコーダ注意、位置ごとのフィードフォワードネットワークである。これらのサブレイヤーには、それぞれの周囲に残差接続があり、その後に層正規化が続く。

この節の前半で述べたように、マスク付きマルチヘッドデコーダ自己注意（1 つ目のサブレイヤー）では、クエリ、キー、値はすべて前のデコーダ層の出力から得られる。系列変換モデルを学習するとき、出力系列のすべての位置（時間ステップ）のトークンは既知である。しかし、予測時には出力系列はトークンごとに生成される。したがって、デコーダの任意の時間ステップでは生成済みトークンのみをデコーダ自己注意に使うことができる。デコーダの自己回帰性を保つために、マスク付き自己注意は `dec_valid_lens` を指定し、各クエリがデコーダ内のクエリ位置までのすべての位置にのみ注意を向けるようにする。

```
class TransformerDecoderBlock(nn.Module):
    # The i-th block in the Transformer decoder
    def __init__(self, num_hiddens, ffn_num_hiddens, num_heads, dropout, i):
        super().__init__()
        self.i = i
        self.attention1 = d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                 dropout)
        self.addnorm1 = AddNorm(num_hiddens, dropout)
        self.attention2 = d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                 dropout)
        self.addnorm2 = AddNorm(num_hiddens, dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm3 = AddNorm(num_hiddens, dropout)

    def forward(self, X, state):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # During training, all the tokens of any output sequence are processed
```

(continues on next page)

(continued from previous page)

```
        # at the same time, so state[2][self.i] is None as initialized. When
        # decoding any output sequence token by token during prediction,
        # state[2][self.i] contains representations of the decoded output at
        # the i-th block up to the current time step
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = torch.cat((state[2][self.i], X), dim=1)
        state[2][self.i] = key_values
        if self.training:
            batch_size, num_steps, _ = X.shape
            # Shape of dec_valid_lens: (batch_size, num_steps), where every
            # row is [1, 2, ..., num_steps]
            dec_valid_lens = torch.arange(
                1, num_steps + 1, device=X.device).repeat(batch_size, 1)
        else:
            dec_valid_lens = None
        # Self-attention
        X2 = self.attention1(X, key_values, key_values, dec_valid_lens)
        Y = self.addnorm1(X, X2)
        # Encoder-decoder attention. Shape of enc_outputs:
        # (batch_size, num_steps, num_hiddens)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens)
        Z = self.addnorm2(Y, Y2)
        return self.addnorm3(Z, self.ffn(Z)), state
```

```
class TransformerDecoderBlock(nn.Block):
    # The i-th block in the Transformer decoder
    def __init__(self, num_hiddens, ffn_num_hiddens, num_heads, dropout, i):
        super().__init__()
        self.i = i
        self.attention1 = d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                 dropout)
        self.addnorm1 = AddNorm(dropout)
        self.attention2 = d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                 dropout)
        self.addnorm2 = AddNorm(dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm3 = AddNorm(dropout)

    def forward(self, X, state):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # During training, all the tokens of any output sequence are processed
        # at the same time, so state[2][self.i] is None as initialized. When
        # decoding any output sequence token by token during prediction,
        # state[2][self.i] contains representations of the decoded output at
        # the i-th block up to the current time step
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = np.concatenate((state[2][self.i], X), axis=1)
        state[2][self.i] = key_values

        if autograd.is_training():
```

(continues on next page)

(continued from previous page)

```
            batch_size, num_steps, _ = X.shape
            # Shape of dec_valid_lens: (batch_size, num_steps), where every
            # row is [1, 2, ..., num_steps]
            dec_valid_lens = np.tile(np.arange(1, num_steps + 1, ctx=X.ctx),
                                     (batch_size, 1))
        else:
            dec_valid_lens = None
        # Self-attention
        X2 = self.attention1(X, key_values, key_values, dec_valid_lens)
        Y = self.addnorm1(X, X2)
        # Encoder-decoder attention. Shape of enc_outputs:
        # (batch_size, num_steps, num_hiddens)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens)
        Z = self.addnorm2(Y, Y2)
        return self.addnorm3(Z, self.ffn(Z)), state
```

```
class TransformerDecoderBlock(nn.Module):
    # The i-th block in the Transformer decoder
    num_hiddens: int
    ffn_num_hiddens: int
    num_heads: int
    dropout: float
    i: int

    def setup(self):
        self.attention1 = d2l.MultiHeadAttention(self.num_hiddens,
                                                 self.num_heads,
                                                 self.dropout)
        self.addnorm1 = AddNorm(self.dropout)
        self.attention2 = d2l.MultiHeadAttention(self.num_hiddens,
                                                 self.num_heads,
                                                 self.dropout)
        self.addnorm2 = AddNorm(self.dropout)
        self.ffn = PositionWiseFFN(self.ffn_num_hiddens, self.num_hiddens)
        self.addnorm3 = AddNorm(self.dropout)

    def __call__(self, X, state, training=False):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # During training, all the tokens of any output sequence are processed
        # at the same time, so state[2][self.i] is None as initialized. When
        # decoding any output sequence token by token during prediction,
        # state[2][self.i] contains representations of the decoded output at
        # the i-th block up to the current time step
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = jnp.concatenate((state[2][self.i], X), axis=1)
        state[2][self.i] = key_values
        if training:
            batch_size, num_steps, _ = X.shape
            # Shape of dec_valid_lens: (batch_size, num_steps), where every
            # row is [1, 2, ..., num_steps]
            dec_valid_lens = jnp.tile(jnp.arange(1, num_steps + 1),
                                      (batch_size, 1))
```

(continues on next page)

(continued from previous page)

```
        else:
            dec_valid_lens = None
        # Self-attention
        X2, attention_w1 = self.attention1(X, key_values, key_values,
                                           dec_valid_lens, training=training)
        Y = self.addnorm1(X, X2, training=training)
        # Encoder-decoder attention. Shape of enc_outputs:
        # (batch_size, num_steps, num_hiddens)
        Y2, attention_w2 = self.attention2(Y, enc_outputs, enc_outputs,
                                           enc_valid_lens, training=training)
        Z = self.addnorm2(Y, Y2, training=training)
        return self.addnorm3(Z, self.ffn(Z), training=training), state,␣
↪attention_w1, attention_w2
```

```
class TransformerDecoderBlock(tf.keras.layers.Layer):
    # The i-th block in the Transformer decoder
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_hiddens, num_heads, dropout, i):
        super().__init__()
        self.i = i
        self.attention1 = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.attention2 = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm2 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm3 = AddNorm(norm_shape, dropout)

    def call(self, X, state, **kwargs):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # During training, all the tokens of any output sequence are processed
        # at the same time, so state[2][self.i] is None as initialized. When
        # decoding any output sequence token by token during prediction,
        # state[2][self.i] contains representations of the decoded output at
        # the i-th block up to the current time step
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = tf.concat((state[2][self.i], X), axis=1)
        state[2][self.i] = key_values
        if kwargs["training"]:
            batch_size, num_steps, _ = X.shape
            # Shape of dec_valid_lens: (batch_size, num_steps), where every
            # row is [1, 2, ..., num_steps]
            dec_valid_lens = tf.repeat(
                tf.reshape(tf.range(1, num_steps + 1),
                           shape=(-1, num_steps)), repeats=batch_size, axis=0)
        else:
            dec_valid_lens = None
        # Self-attention
        X2 = self.attention1(X, key_values, key_values, dec_valid_lens,
                             **kwargs)
        Y = self.addnorm1(X, X2, **kwargs)
```

(continues on next page)

(continued from previous page)

```
        # Encoder-decoder attention. Shape of enc_outputs:
        # (batch_size, num_steps, num_hiddens)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens,
                             **kwargs)
        Z = self.addnorm2(Y, Y2, **kwargs)
        return self.addnorm3(Z, self.ffn(Z), **kwargs), state
```

エンコーダ–デコーダ注意におけるスケールド・ドット積演算と残差接続における加算演算を容易にするため、デコーダの特徴次元（num_hiddens）はエンコーダのそれと同じである。

```
decoder_blk = TransformerDecoderBlock(24, 48, 8, 0.5, 0)
X = d2l.ones((2, 100, 24))
state = [encoder_blk(X, valid_lens), valid_lens, [None]]
d2l.check_shape(decoder_blk(X, state)[0], X.shape)
```

```
decoder_blk = TransformerDecoderBlock(24, 48, 8, 0.5, 0)
decoder_blk.initialize()
X = np.ones((2, 100, 24))
state = [encoder_blk(X, valid_lens), valid_lens, [None]]
d2l.check_shape(decoder_blk(X, state)[0], X.shape)
```

```
decoder_blk = TransformerDecoderBlock(24, 48, 8, 0.5, 0)
X = d2l.ones((2, 100, 24))
state = [encoder_blk.init_with_output(d2l.get_key(), X, valid_lens)[0][0],
         valid_lens, [None]]
d2l.check_shape(decoder_blk.init_with_output(d2l.get_key(), X, state)[0][0],
                X.shape)
```

```
decoder_blk = TransformerDecoderBlock(24, 24, 24, 24, [1, 2], 48, 8, 0.5, 0)
X = tf.ones((2, 100, 24))
state = [encoder_blk(X, valid_lens), valid_lens, [None]]
d2l.check_shape(decoder_blk(X, state, training=False)[0], X.shape)
```

ここで、num_blks 個の TransformerDecoderBlock インスタンスからなる Transformer デコーダ全体を構成する。最後に、全結合層が vocab_size 個の可能な出力トークンすべてに対する予測を計算する。デコーダ自己注意の重みとエンコーダ–デコーダ注意の重みの両方は、後で可視化できるように保存される。

```
class TransformerDecoder(d2l.AttentionDecoder):
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                 num_blks, dropout):
        super().__init__()
        self.num_hiddens = num_hiddens
        self.num_blks = num_blks
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
```

(continues on next page)

(continued from previous page)

```
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_blks):
            self.blks.add_module("block"+str(i), TransformerDecoderBlock(
                num_hiddens, ffn_num_hiddens, num_heads, dropout, i))
        self.dense = nn.LazyLinear(vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens):
        return [enc_outputs, enc_valid_lens, [None] * self.num_blks]

    def forward(self, X, state):
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self._attention_weights = [[None] * len(self.blks) for _ in range (2)]
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state)
            # Decoder self-attention weights
            self._attention_weights[0][
                i] = blk.attention1.attention.attention_weights
            # Encoder-decoder attention weights
            self._attention_weights[1][
                i] = blk.attention2.attention.attention_weights
        return self.dense(X), state

    @property
    def attention_weights(self):
        return self._attention_weights
```

```
class TransformerDecoder(d2l.AttentionDecoder):
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                 num_blks, dropout):
        super().__init__()
        self.num_hiddens = num_hiddens
        self.num_blks = num_blks
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_blks):
            self.blks.add(TransformerDecoderBlock(
                num_hiddens, ffn_num_hiddens, num_heads, dropout, i))
        self.dense = nn.Dense(vocab_size, flatten=False)
        self.initialize()

    def init_state(self, enc_outputs, enc_valid_lens):
        return [enc_outputs, enc_valid_lens, [None] * self.num_blks]

    def forward(self, X, state):
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self._attention_weights = [[None] * len(self.blks) for _ in range (2)]
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state)
            # Decoder self-attention weights
            self._attention_weights[0][
                i] = blk.attention1.attention.attention_weights
            # Encoder-decoder attention weights
```

(continues on next page)

(continued from previous page)

```
            self._attention_weights[1][
                i] = blk.attention2.attention.attention_weights
        return self.dense(X), state

    @property
    def attention_weights(self):
        return self._attention_weights
```

```
class TransformerDecoder(nn.Module):
    vocab_size: int
    num_hiddens: int
    ffn_num_hiddens: int
    num_heads: int
    num_blks: int
    dropout: float

    def setup(self):
        self.embedding = nn.Embed(self.vocab_size, self.num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(self.num_hiddens,
                                                   self.dropout)
        self.blks = [TransformerDecoderBlock(self.num_hiddens,
                                             self.ffn_num_hiddens,
                                             self.num_heads, self.dropout, i)
                     for i in range(self.num_blks)]
        self.dense = nn.Dense(self.vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens):
        return [enc_outputs, enc_valid_lens, [None] * self.num_blks]

    def __call__(self, X, state, training=False):
        X = self.embedding(X) * jnp.sqrt(jnp.float32(self.num_hiddens))
        X = self.pos_encoding(X, training=training)
        attention_weights = [[None] * len(self.blks) for _ in range(2)]
        for i, blk in enumerate(self.blks):
            X, state, attention_w1, attention_w2 = blk(X, state,
                                                       training=training)
            # Decoder self-attention weights
            attention_weights[0][i] = attention_w1
            # Encoder-decoder attention weights
            attention_weights[1][i] = attention_w2
        # Flax sow API is used to capture intermediate variables
        self.sow('intermediates', 'dec_attention_weights', attention_weights)
        return self.dense(X), state
```

```
class TransformerDecoder(d2l.AttentionDecoder):
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_hiddens, num_heads,
                 num_blks, dropout):
        super().__init__()
        self.num_hiddens = num_hiddens
        self.num_blks = num_blks
        self.embedding = tf.keras.layers.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
```

(continues on next page)

(continued from previous page)

```
        self.blks = [TransformerDecoderBlock(
            key_size, query_size, value_size, num_hiddens, norm_shape,
            ffn_num_hiddens, num_heads, dropout, i)
                     for i in range(num_blks)]
        self.dense = tf.keras.layers.Dense(vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens):
        return [enc_outputs, enc_valid_lens, [None] * self.num_blks]

    def call(self, X, state, **kwargs):
        X = self.pos_encoding(self.embedding(X) * tf.math.sqrt(
            tf.cast(self.num_hiddens, dtype=tf.float32)), **kwargs)
        # 2 attention layers in decoder
        self._attention_weights = [[None] * len(self.blks) for _ in range(2)]
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state, **kwargs)
            # Decoder self-attention weights
            self._attention_weights[0][i] = (
                blk.attention1.attention.attention_weights)
            # Encoder-decoder attention weights
            self._attention_weights[1][i] = (
                blk.attention2.attention.attention_weights)
        return self.dense(X), state

    @property
    def attention_weights(self):
        return self._attention_weights
```

14.7.6 学習

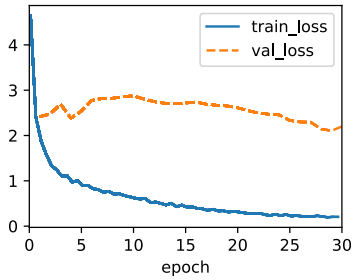
Transformer アーキテクチャに従ってエンコーダ–デコーダモデルを実装しよう。ここでは、Transformer エンコーダと Transformer デコーダの両方が 4 ヘッド注意を用いた 2 層構成であるとする。Section 13.7.6 と同様に、英仏機械翻訳データセット上で系列変換学習のために Transformer モデルを学習する。

```
data = d2l.MTFraEng(batch_size=128)
num_hiddens, num_blks, dropout = 256, 2, 0.2
ffn_num_hiddens, num_heads = 64, 4
if tab.selected('tensorflow'):
    key_size, query_size, value_size = 256, 256, 256
    norm_shape = [2]
if tab.selected('pytorch', 'mxnet', 'jax'):
    encoder = TransformerEncoder(
        len(data.src_vocab), num_hiddens, ffn_num_hiddens, num_heads,
        num_blks, dropout)
    decoder = TransformerDecoder(
        len(data.tgt_vocab), num_hiddens, ffn_num_hiddens, num_heads,
        num_blks, dropout)
if tab.selected('mxnet', 'pytorch'):
    model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.001)
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
```

(continues on next page)

(continued from previous page)

```
if tab.selected('jax'):
    model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                        lr=0.001, training=True)
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1, num_gpus=1)
if tab.selected('tensorflow'):
    with d2l.try_gpu():
        encoder = TransformerEncoder(
            len(data.src_vocab), key_size, query_size, value_size, num_hiddens,
            norm_shape, ffn_num_hiddens, num_heads, num_blks, dropout)
        decoder = TransformerDecoder(
            len(data.tgt_vocab), key_size, query_size, value_size, num_hiddens,
            norm_shape, ffn_num_hiddens, num_heads, num_blks, dropout)
        model = d2l.Seq2Seq(encoder, decoder, tgt_pad=data.tgt_vocab['<pad>'],
                            lr=0.001)
    trainer = d2l.Trainer(max_epochs=30, gradient_clip_val=1)
trainer.fit(model, data)
```



学習後、Transformer モデルを用いていくつかの英語文をフランス語に翻訳し、その BLEU スコアを計算する。

```
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
if tab.selected('pytorch', 'mxnet', 'tensorflow'):
    preds, _ = model.predict_step(
        data.build(engs, fras), d2l.try_gpu(), data.num_steps)
if tab.selected('jax'):
    preds, _ = model.predict_step(
        trainer.state.params, data.build(engs, fras), data.num_steps)
for en, fr, p in zip(engs, fras, preds):
    translation = []
    for token in data.tgt_vocab.to_tokens(p):
        if token == '<eos>':
            break
        translation.append(token)
    print(f'{en} => {translation}, bleu,'
          f'{d2l.bleu(" ".join(translation), fr, k=2):.3f}')
```

```
go . => ['va', '!'], bleu,1.000
i lost . => ["j'ai", 'perdu', '.'], bleu,1.000
```

(continues on next page)

(continued from previous page)

```
he's calm . => ['<unk>', '.'], bleu,0.000
i'm home . => ['je', 'suis', 'chez', 'moi', '.'], bleu,1.000
```

最後の英語文をフランス語に翻訳するときの Transformer の注意重みを可視化する。エンコーダ自己注意重みの形状は（エンコーダ層数、注意ヘッド数、num_steps またはクエリ数、num_steps またはキー・値ペア数）である。

```
_, dec_attention_weights = model.predict_step(
    data.build([engs[-1]], [fras[-1]]), d2l.try_gpu(), data.num_steps, True)
enc_attention_weights = d2l.concat(model.encoder.attention_weights, 0)
shape = (num_blks, num_heads, -1, data.num_steps)
enc_attention_weights = d2l.reshape(enc_attention_weights, shape)
d2l.check_shape(enc_attention_weights,
                (num_blks, num_heads, data.num_steps, data.num_steps))
```

```
_, dec_attention_weights = model.predict_step(
    data.build([engs[-1]], [fras[-1]]), d2l.try_gpu(), data.num_steps, True)
enc_attention_weights = d2l.concat(model.encoder.attention_weights, 0)
shape = (num_blks, num_heads, -1, data.num_steps)
enc_attention_weights = d2l.reshape(enc_attention_weights, shape)
d2l.check_shape(enc_attention_weights,
                (num_blks, num_heads, data.num_steps, data.num_steps))
```

```
_, (dec_attention_weights, enc_attention_weights) = model.predict_step(
    trainer.state.params, data.build([engs[-1]], [fras[-1]]),
    data.num_steps, True)
enc_attention_weights = d2l.concat(enc_attention_weights, 0)
shape = (num_blks, num_heads, -1, data.num_steps)
enc_attention_weights = d2l.reshape(enc_attention_weights, shape)
d2l.check_shape(enc_attention_weights,
                (num_blks, num_heads, data.num_steps, data.num_steps))
```

```
_, dec_attention_weights = model.predict_step(
    data.build([engs[-1]], [fras[-1]]), d2l.try_gpu(), data.num_steps, True)
enc_attention_weights = d2l.concat(model.encoder.attention_weights, 0)
shape = (num_blks, num_heads, -1, data.num_steps)
enc_attention_weights = d2l.reshape(enc_attention_weights, shape)
d2l.check_shape(enc_attention_weights,
                (num_blks, num_heads, data.num_steps, data.num_steps))
```

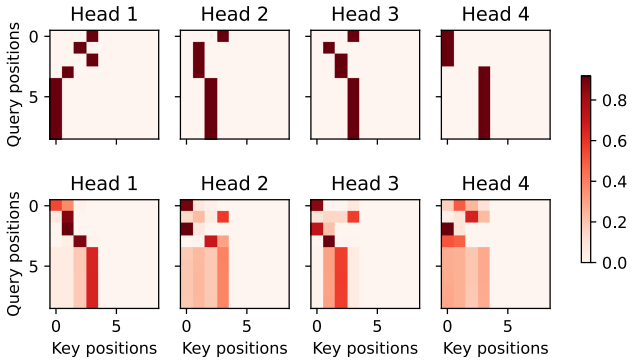
エンコーダ自己注意では、クエリとキーの両方が同じ入力系列から来る。パディングトークンは意味を持たないため、入力系列の有効長が指定されていれば、どのクエリもパディングトークンの位置には注意を向けない。以下では、マルチヘッド注意重みの 2 層を行ごとに示す。各ヘッドは、クエリ、キー、値の別々の表現部分空間に基づいて独立に注意を向ける。

```
d2l.show_heatmaps(
```

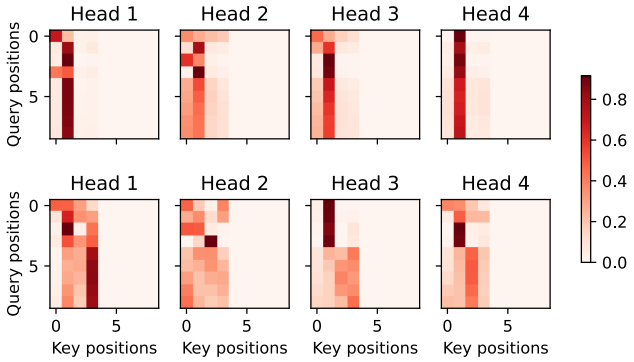
(continues on next page)

(continued from previous page)

```
    enc_attention_weights.cpu(), xlabel='Key positions',
    ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
    figsize=(7, 3.5))
```



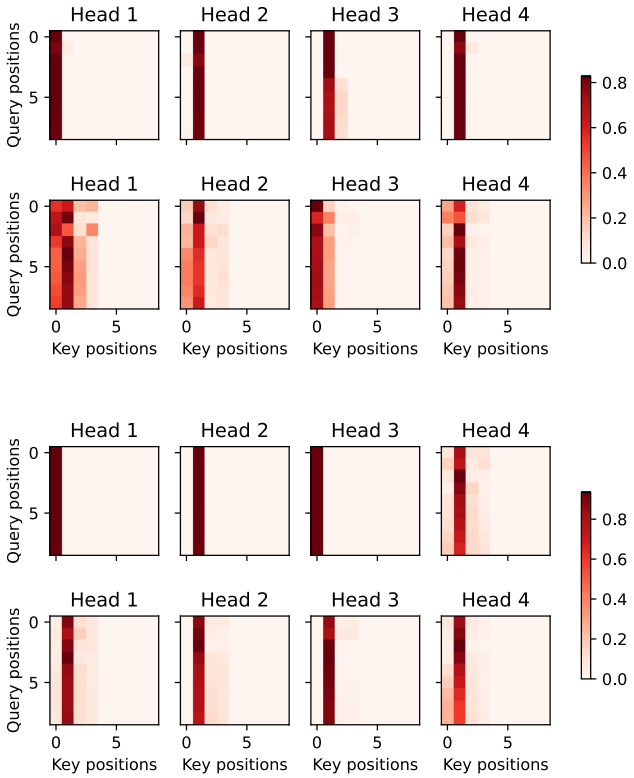

```
d2l.show_heatmaps(
    enc_attention_weights, xlabel='Key positions', ylabel='Query positions',
    titles=['Head %d' % i for i in range(1, 5)], figsize=(7, 3.5))
```




```
d2l.show_heatmaps(
    enc_attention_weights, xlabel='Key positions', ylabel='Query positions',
    titles=['Head %d' % i for i in range(1, 5)], figsize=(7, 3.5))
```

```
d2l.show_heatmaps(
    enc_attention_weights, xlabel='Key positions', ylabel='Query positions',
    titles=['Head %d' % i for i in range(1, 5)], figsize=(7, 3.5))
```

デコーダ自己注意重みとエンコーダ–デコーダ注意重みを可視化するには、さらにデータ操作が必要である。たとえば、マスクされた注意重みを 0 で埋める。なお、デコーダ自己注意重みとエンコーダ–デコーダ注意重みはどちらも同じク



エリを持つ。すなわち、系列開始トークンの後に出力トークン、場合によっては系列終了トークンが続く。

```
dec_attention_weights_2d = [head[0].tolist()
                            for step in dec_attention_weights
                            for attn in step for blk in attn for head in blk]
dec_attention_weights_filled = d2l.tensor(
    pd.DataFrame(dec_attention_weights_2d).fillna(0.0).values)
shape = (-1, 2, num_blks, num_heads, data.num_steps)
dec_attention_weights = d2l.reshape(dec_attention_weights_filled, shape)
dec_self_attention_weights, dec_inter_attention_weights = \
    dec_attention_weights.permute(1, 2, 3, 0, 4)
```

```
dec_attention_weights_2d = [d2l.tensor(head[0]).tolist()
                            for step in dec_attention_weights
                            for attn in step for blk in attn for head in blk]
dec_attention_weights_filled = d2l.tensor(
    pd.DataFrame(dec_attention_weights_2d).fillna(0.0).values)
dec_attention_weights = d2l.reshape(dec_attention_weights_filled, (
    -1, 2, num_blks, num_heads, data.num_steps))
dec_self_attention_weights, dec_inter_attention_weights = \
    dec_attention_weights.transpose(1, 2, 3, 0, 4)
```

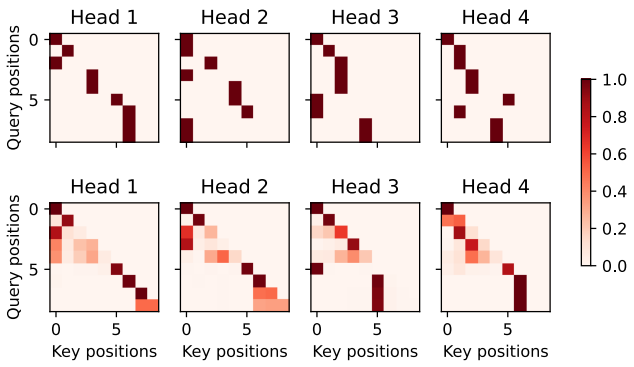
```
dec_attention_weights_2d = [head[0].tolist() for step in dec_attention_weights
                            for attn in step
                            for blk in attn for head in blk]
dec_attention_weights_filled = d2l.tensor(
    pd.DataFrame(dec_attention_weights_2d).fillna(0.0).values)
dec_attention_weights = dec_attention_weights_filled.reshape(
    (-1, 2, num_blks, num_heads, data.num_steps))
dec_self_attention_weights, dec_inter_attention_weights = \
    dec_attention_weights.transpose(1, 2, 3, 0, 4)
```

```
dec_attention_weights_2d = [head[0] for step in dec_attention_weights
                            for attn in step
                            for blk in attn for head in blk]
dec_attention_weights_filled = tf.convert_to_tensor(
    np.asarray(pd.DataFrame(dec_attention_weights_2d).fillna(
        0.0).values).astype(np.float32))
dec_attention_weights = tf.reshape(dec_attention_weights_filled, shape=(
    -1, 2, num_blks, num_heads, data.num_steps))
dec_self_attention_weights, dec_inter_attention_weights = tf.transpose(
    dec_attention_weights, perm=(1, 2, 3, 0, 4))
```

```
d2l.check_shape(dec_self_attention_weights,
                (num_blks, num_heads, data.num_steps, data.num_steps))
d2l.check_shape(dec_inter_attention_weights,
                (num_blks, num_heads, data.num_steps, data.num_steps))
```

デコーダ自己注意の自己回帰性のため、どのクエリもクエリ位置より後のキー・値ペアには注意を向けない。

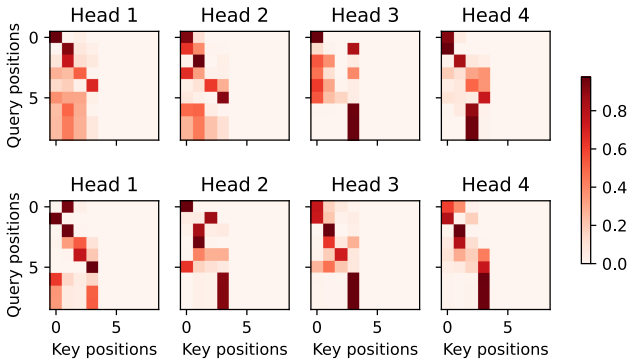
```
d2l.show_heatmaps(
    dec_self_attention_weights[:, :, :, :],
    xlabel='Key positions', ylabel='Query positions',
    titles=['Head %d' % i for i in range(1, 5)], figsize=(7, 3.5))
```



エンコーダ自己注意の場合と同様に、入力系列の有効長を指定することで、出

力系列からのどのクエリも入力系列中のパディングトークンには注意を向けない。

```
d2l.show_heatmaps(
    dec_inter_attention_weights, xlabel='Key positions',
    ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
    figsize=(7, 3.5))
```



Transformer アーキテクチャはもともと系列変換学習のために提案されたが、本書の後半で見るように、Transformer エンコーダまたは Transformer デコーダのいずれか一方がさまざまな深層学習タスクで個別に使われることが多い。

14.7.7 要約

Transformer はエンコーダ–デコーダアーキテクチャの一例であるが、実際にはエンコーダまたはデコーダのどちらか一方だけを使うこともできる。Transformer アーキテクチャでは、マルチヘッド自己注意が入力系列と出力系列の表現に用いられるが、デコーダではマスク付き版によって自己回帰性を保たなければならない。Transformer における残差接続と層正規化は、非常に深いモデルを学習するうえで重要である。Transformer モデルの位置ごとのフィードフォワードネットワークは、同じ MLP を用いてすべての系列位置の表現を変換する。

14.7.8 演習

1. 実験でより深い Transformer を学習せよ。学習速度と翻訳性能にどのような影響があるか。
2. Transformer でスケールド・ドット積注意を加法注意に置き換えるのはよい考えか。なぜか。
3. 言語モデル化では、Transformer のエンコーダ、デコーダ、あるいは両方のどれを使うべきか。この方法をどのように設計するか。
4. 入力系列が非常に長い場合、Transformer はどのような課題に直面するか。なぜか。

5. Transformer の計算効率とメモリ効率をどのように改善するか。ヒント：Tay *et al.* (2020) のサーベイ論文を参照されたい。

14.8 画像向け Transformer

Transformer アーキテクチャは当初、機械翻訳に焦点を当てた系列変換学習のために提案された。その後、Transformer はさまざまな自然言語処理タスクにおける第一選択のモデルとして台頭した (Brown *et al.*, 2020, Devlin *et al.*, 2018, Radford *et al.*, 2018, Radford *et al.*, 2019, Raffel *et al.*, 2020)。しかし、コンピュータビジョンの分野では、支配的なアーキテクチャは依然として CNN のままであった（Section 11）。当然ながら、研究者たちは Transformer モデルを画像データに適用することでより良い性能が得られるのではないかと考え始めた。この問いは、コンピュータビジョンコミュニティに大きな関心を呼び起こした。最近では、Ramachandran *et al.* (2019) が畳み込みを自己注意で置き換える方式を提案した。しかし、注意機構に特殊なパターンを用いるため、ハードウェアアクセラレータ上でモデルを大規模化しにくい。その後、Cordonnier *et al.* (2020) は理論的に、自己注意が畳み込みと同様に振る舞うよう学習できることを証明した。実証的には、画像から 2×2 のパッチを入力として取り出したが、パッチサイズが小さいため、このモデルは低解像度の画像データにしか適用できない。

パッチサイズに特別な制約を設けずに、*vision Transformers*（ViT）は画像からパッチを抽出し、それらを Transformer エンコーダに入力してグローバルな表現を得る。そして最終的に、その表現を分類用に変換する (Dosovitskiy *et al.*, 2021)。特筆すべきは、Transformer は CNN よりもスケーラビリティに優れていることである。より大きなデータセットでより大規模なモデルを学習すると、vision Transformer は ResNet を大きく上回る。自然言語処理におけるネットワークアーキテクチャ設計の潮流と同様に、Transformer はコンピュータビジョンにおいてもゲームチェンジャーとなった。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
img_size, patch_size = 96, 16
num_hiddens, mlp_num_hiddens, num_heads, num_blks = 512, 2048, 8, 2
emb_dropout, blk_dropout, lr = 0.1, 0.1, 0.1
model = ViT(img_size, patch_size, num_hiddens, mlp_num_hiddens, num_heads,
            num_blks, emb_dropout, blk_dropout, lr)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(img_size, img_size))
trainer.fit(model, data)
```

```
from d2l import jax as d2l
from flax import linen as nn
import jax
from jax import numpy as jnp
```

```
img_size, patch_size = 96, 16
num_hiddens, mlp_num_hiddens, num_heads, num_blks = 512, 2048, 8, 2
emb_dropout, blk_dropout, lr = 0.1, 0.1, 0.1
model = ViT(img_size, patch_size, num_hiddens, mlp_num_hiddens, num_heads,
            num_blks, emb_dropout, blk_dropout, lr)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(img_size, img_size))
trainer.fit(model, data)
```

14.8.1 モデル

Fig. 14.8.1 は vision Transformer のモデルアーキテクチャを示している。このアーキテクチャは、画像をパッチ化する stem、多層 Transformer エンコーダに基づく body、そしてグローバル表現を出力ラベルへ変換する head から構成される。

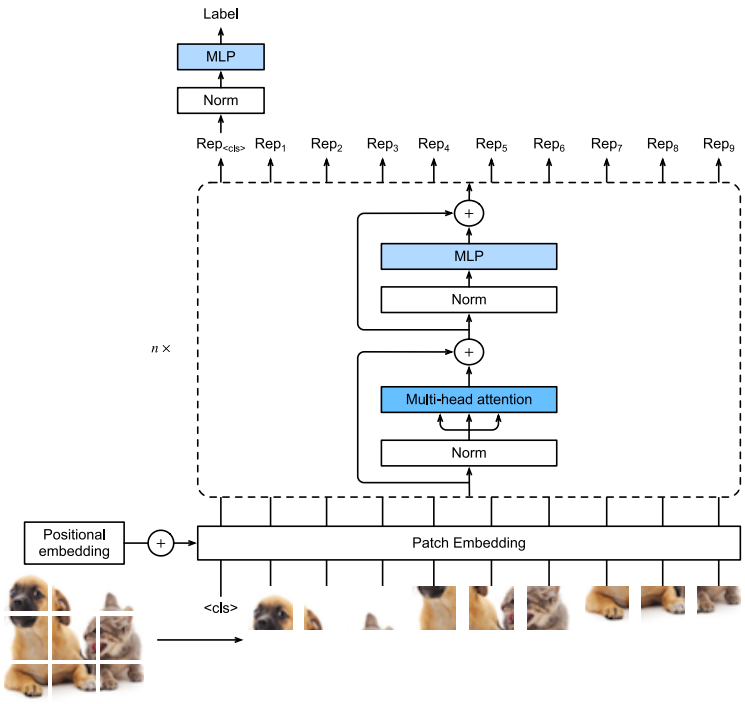


Fig. 14.8.1 The vision Transformer architecture. In this example, an image is split into nine patches. A special “<cls>”token and the nine flattened image patches are transformed via patch embedding and n Transformer encoder blocks into ten representations, respectively. The “<cls>”representation is further transformed into the output label.

高さ h、幅 w、チャネル数 c をもつ入力画像を考える。パッチの高さと幅をともに p とすると、画像は $m = hw/p^2$ 個のパッチ列に分割され、各パッチは長さ cp^2 のベクトルに平坦化される。このようにして、画像パッチは Transformer エンコーダによってテキスト系列中のトークンと同様に扱うことができる。特別な“<cls>”（class）トークンと m 個の平坦化された画像パッチは線形射影されて $m+1$ 個のベクトル列となり、学習可能な位置埋め込みが加算される。多層 Transformer エンコーダは $m+1$ 個の入力ベクトルを同じ長さの $m+1$ 個の出力ベクトル表現へ変換する。これは Fig. 14.7.1 における元の Transformer エンコーダとまったく同じように動作し、正規化の位置だけが異なる。“<cls>”トークンは自己注意を通じてすべての画像パッチに注意を向けるため（Fig. 14.6.1 を参照）、Transformer エンコーダ出力におけるその表現はさらに出力ラベルへ変換される。

14.8.2 パッチ埋め込み

vision Transformer を実装するには、まず Fig. 14.8.1 のパッチ埋め込みから始めよう。画像をパッチに分割し、それらの平坦化されたパッチを線形射影する操作は、カーネルサイズとストライドサイズの両方をパッチサイズに設定した 1 つの畳み込み演算として簡略化できる。

```
class PatchEmbedding(nn.Module):
    def __init__(self, img_size=96, patch_size=16, num_hiddens=512):
        super().__init__()
        def _make_tuple(x):
            if not isinstance(x, (list, tuple)):
                return (x, x)
            return x
        img_size, patch_size = _make_tuple(img_size), _make_tuple(patch_size)
        self.num_patches = (img_size[0] // patch_size[0]) * (
            img_size[1] // patch_size[1])
        self.conv = nn.LazyConv2d(num_hiddens, kernel_size=patch_size,
                                  stride=patch_size)

    def forward(self, X):
        # Output shape: (batch size, no. of patches, no. of channels)
        return self.conv(X).flatten(2).transpose(1, 2)
```

```
class PatchEmbedding(nn.Module):
    img_size: int = 96
    patch_size: int = 16
    num_hiddens: int = 512

    def setup(self):
        def _make_tuple(x):
            if not isinstance(x, (list, tuple)):
                return (x, x)
            return x
        img_size, patch_size = _make_tuple(self.img_size), _make_tuple(self.
↪patch_size)
        self.num_patches = (img_size[0] // patch_size[0]) * (
```

(continues on next page)

(continued from previous page)

```
            img_size[1] // patch_size[1])
        self.conv = nn.Conv(self.num_hiddens, kernel_size=patch_size,
                            strides=patch_size, padding='SAME')

    def __call__(self, X):
        # Output shape: (batch size, no. of patches, no. of channels)
        X = self.conv(X)
        return X.reshape((X.shape[0], -1, X.shape[3]))
```

次の例では、高さと幅が `img_size` の画像を入力として、パッチ埋め込みは `(img_size//patch_size)**2` 個のパッチを出力し、それらは長さ `num_hiddens` のベクトルへ線形射影される。

```
img_size, patch_size, num_hiddens, batch_size = 96, 16, 512, 4
patch_emb = PatchEmbedding(img_size, patch_size, num_hiddens)
X = d2l.zeros(batch_size, 3, img_size, img_size)
d2l.check_shape(patch_emb(X),
                (batch_size, (img_size//patch_size)**2, num_hiddens))
```

```
img_size, patch_size, num_hiddens, batch_size = 96, 16, 512, 4
patch_emb = PatchEmbedding(img_size, patch_size, num_hiddens)
X = d2l.zeros((batch_size, img_size, img_size, 3))
output, _ = patch_emb.init_with_output(d2l.get_key(), X)
d2l.check_shape(output, (batch_size, (img_size//patch_size)**2, num_hiddens))
```

14.8.3 Vision Transformer エンコーダ

vision Transformer エンコーダの MLP は、元の Transformer エンコーダの位置ごとの FFN とは少し異なる（Section 14.7.2 を参照）。第一に、ここでは活性化関数としてガウス誤差線形ユニット（GELU）を用いる。これは ReLU のより滑らかな版とみなせる (Hendrycks and Gimpel, 2016)。第二に、正則化のために、MLP 内の各全結合層の出力にドロップアウトを適用する。

```
class ViTMLP(nn.Module):
    def __init__(self, mlp_num_hiddens, mlp_num_outputs, dropout=0.5):
        super().__init__()
        self.dense1 = nn.LazyLinear(mlp_num_hiddens)
        self.gelu = nn.GELU()
        self.dropout1 = nn.Dropout(dropout)
        self.dense2 = nn.LazyLinear(mlp_num_outputs)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, x):
        return self.dropout2(self.dense2(self.dropout1(self.gelu(
            self.dense1(x)))))
```

```
class ViTMLP(nn.Module):
    mlp_num_hiddens: int
```

(continues on next page)

(continued from previous page)

```
    mlp_num_outputs: int
    dropout: float = 0.5

    @nn.compact
    def __call__(self, x, training=False):
        x = nn.Dense(self.mlp_num_hiddens)(x)
        x = nn.gelu(x)
        x = nn.Dropout(self.dropout, deterministic=not training)(x)
        x = nn.Dense(self.mlp_num_outputs)(x)
        x = nn.Dropout(self.dropout, deterministic=not training)(x)
        return x
```

vision Transformer エンコーダブロックの実装は、Fig. 14.8.1 における事前正規化の設計に従っている。ここでは、正規化はマルチヘッド注意または MLP の直前に適用される。Fig. 14.7.1 の「add & norm」のような事後正規化では、正規化は残差接続の直後に置かれるのに対し、事前正規化は Transformer の学習をより効果的または効率的にする (Baevski and Auli, 2018, Wang *et al.*, 2019, Xiong *et al.*, 2020)。

```
class ViTBlock(nn.Module):
    def __init__(self, num_hiddens, norm_shape, mlp_num_hiddens,
                 num_heads, dropout, use_bias=False):
        super().__init__()
        self.ln1 = nn.LayerNorm(norm_shape)
        self.attention = d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                dropout, use_bias)
        self.ln2 = nn.LayerNorm(norm_shape)
        self.mlp = ViTMLP(mlp_num_hiddens, num_hiddens, dropout)

    def forward(self, X, valid_lens=None):
        X = X + self.attention(*([self.ln1(X)] * 3), valid_lens)
        return X + self.mlp(self.ln2(X))
```

```
class ViTBlock(nn.Module):
    num_hiddens: int
    mlp_num_hiddens: int
    num_heads: int
    dropout: float
    use_bias: bool = False

    def setup(self):
        self.attention = d2l.MultiHeadAttention(self.num_hiddens, self.num_
↪heads,
                                                self.dropout, self.use_bias)
        self.mlp = ViTMLP(self.mlp_num_hiddens, self.num_hiddens, self.dropout)

    @nn.compact
    def __call__(self, X, valid_lens=None, training=False):
        X = X + self.attention(*([nn.LayerNorm()(X)] * 3),
                               valid_lens, training=training)[0]
        return X + self.mlp(nn.LayerNorm()(X), training=training)
```

Section 14.7.4 と同様に、vision Transformer エンコーダブロックは入力の形状を変えない。

```
X = d2l.ones((2, 100, 24))
encoder_blk = ViTBlock(24, 24, 48, 8, 0.5)
encoder_blk.eval()
d2l.check_shape(encoder_blk(X), X.shape)
```

```
X = d2l.ones((2, 100, 24))
encoder_blk = ViTBlock(24, 48, 8, 0.5)
d2l.check_shape(encoder_blk.init_with_output(d2l.get_key(), X)[0], X.shape)
```

14.8.4 全体をまとめる

以下の vision Transformer の順伝播は単純である。まず、入力画像は `PatchEmbedding` インスタンスに与えられ、その出力は“<cls>”トークン埋め込みと連結される。それらはドロップアウトの前に、学習可能な位置埋め込みと加算される。次に、その出力は `ViTBlock` クラスのインスタンスを `num_blks` 個積み重ねた Transformer エンコーダに入力される。最後に、“<cls>”トークンの表現がネットワークの head によって射影される。

```
class ViT(d2l.Classifier):
    """Vision Transformer."""
    def __init__(self, img_size, patch_size, num_hiddens, mlp_num_hiddens,
                 num_heads, num_blks, emb_dropout, blk_dropout, lr=0.1,
                 use_bias=False, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.patch_embedding = PatchEmbedding(
            img_size, patch_size, num_hiddens)
        self.cls_token = nn.Parameter(d2l.zeros(1, 1, num_hiddens))
        num_steps = self.patch_embedding.num_patches + 1  # Add the cls token
        # Positional embeddings are learnable
        self.pos_embedding = nn.Parameter(
            torch.randn(1, num_steps, num_hiddens))
        self.dropout = nn.Dropout(emb_dropout)
        self.blks = nn.Sequential()
        for i in range(num_blks):
            self.blks.add_module(f"{i}", ViTBlock(
                num_hiddens, num_hiddens, mlp_num_hiddens,
                num_heads, blk_dropout, use_bias))
        self.head = nn.Sequential(nn.LayerNorm(num_hiddens),
                                  nn.Linear(num_hiddens, num_classes))

    def forward(self, X):
        X = self.patch_embedding(X)
        X = d2l.concat((self.cls_token.expand(X.shape[0], -1, -1), X), 1)
        X = self.dropout(X + self.pos_embedding)
        for blk in self.blks:
            X = blk(X)
        return self.head(X[:, 0])
```

```
class ViT(d2l.Classifier):
    """Vision Transformer."""
    img_size: int
    patch_size: int
    num_hiddens: int
    mlp_num_hiddens: int
    num_heads: int
    num_blks: int
    emb_dropout: float
    blk_dropout: float
    lr: float = 0.1
    use_bias: bool = False
    num_classes: int = 10
    training: bool = False

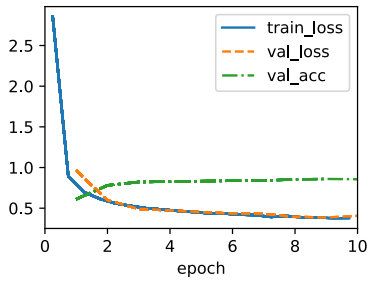
    def setup(self):
        self.patch_embedding = PatchEmbedding(self.img_size, self.patch_size,
                                              self.num_hiddens)
        self.cls_token = self.param('cls_token', nn.initializers.zeros,
                                    (1, 1, self.num_hiddens))
        num_steps = self.patch_embedding.num_patches + 1  # Add the cls token
        # Positional embeddings are learnable
        self.pos_embedding = self.param('pos_embed', nn.initializers.normal(),
                                        (1, num_steps, self.num_hiddens))
        self.blks = [ViTBlock(self.num_hiddens, self.mlp_num_hiddens,
                              self.num_heads, self.blk_dropout, self.use_bias)
                     for _ in range(self.num_blks)]
        self.head = nn.Sequential([nn.LayerNorm(), nn.Dense(self.num_classes)])

    @nn.compact
    def __call__(self, X):
        X = self.patch_embedding(X)
        X = d2l.concat((jnp.tile(self.cls_token, (X.shape[0], 1, 1)), X), 1)
        X = nn.Dropout(emb_dropout, deterministic=not self.training)(X + self.
↪pos_embedding)
        for blk in self.blks:
            X = blk(X, training=self.training)
        return self.head(X[:, 0])
```

14.8.5 学習

Fashion-MNIST データセットで vision Transformer を学習するのは、Section 11 で CNN を学習したときと同じである。

```
img_size, patch_size = 96, 16
num_hiddens, mlp_num_hiddens, num_heads, num_blks = 512, 2048, 8, 2
emb_dropout, blk_dropout, lr = 0.1, 0.1, 0.1
model = ViT(img_size, patch_size, num_hiddens, mlp_num_hiddens, num_heads,
            num_blks, emb_dropout, blk_dropout, lr)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(img_size, img_size))
trainer.fit(model, data)
```



14.8.6 要約と考察

Fashion-MNIST のような小規模データセットでは、実装した vision Transformer が Section 11.6 の ResNet を上回らないことに気づいたかもしれない。同様の観察は、ImageNet データセット（120 万枚の画像）でも成り立つ。これは、Transformer には畳み込みにおける有用な帰納バイアス、たとえば平行移動不変性や局所性（Section 10.1）が 欠けているためである。しかし、より大きなデータセット（たとえば 3 億枚の画像）でより大規模なモデルを学習すると状況は変わり、その場合、vision Transformer は画像分類で ResNet を大きく上回り、スケーラビリティにおける Transformer の本質的な優位性を示している (Dosovitskiy *et al.*, 2021)。vision Transformer の導入は、画像データをモデル化するためのネットワーク設計の潮流を変えた。その後すぐに、DeiT のデータ効率の高い学習戦略によって ImageNet データセットで有効であることが示された (Touvron *et al.*, 2021)。しかし、自己注意の二次計算量（Section 14.6）のため、Transformer アーキテクチャは高解像度画像にはあまり適していない。コンピュータビジョンにおける汎用バックボーンネットワークを目指して、Swin Transformer は画像サイズに対する二次的な計算複雑性を（Section 14.6.2）解消し、畳み込みに似た事前知識を再導入した。その結果、Transformer の適用範囲は画像分類を超えたさまざまなコンピュータビジョンタスクへと広がり、最先端の結果を達成している (Liu *et al.*, 2021)。

14.8.7 演習

1. `img_size` の値は学習時間にどのように影響するか。

2. “<cls>”トークン表現を出力へ射影する代わりに、平均化したパッチ表現をどのように射影するか。これを実装し、精度への影響を調べよ。

3. ハイパーパラメータを調整して、vision Transformer の精度を改善できるか。

14.9 Transformer による大規模事前学習

これまでの画像分類と機械翻訳の実験では、モデルは入力–出力の例を含むデータセットで特定のタスクを実行するために ゼロから学習されてきた。たとえば、Transformer は英語–フランス語の対で学習された (Section 14.7)。これにより、このモデルは入力された英語テキストをフランス語に翻訳できる。その結果、各モデルはデータ分布のわずかな変化にも敏感な 特化型の専門家になる (Section 7.7)。より汎化性能の高いモデル、あるいは適応の有無にかかわらず複数のタスクを実行できるより有能な 汎用モデルを得るために、大規模データでモデルを 事前学習することがますます一般的になっている。

事前学習のためのデータが大きくなると、Transformer アーキテクチャはモデルサイズと学習計算量を増やすことでより良い性能を示し、優れた スケーリング挙動を示す。具体的には、Transformer ベースの言語モデルの性能は、モデルパラメータ数、学習トークン数、学習計算量に対してべき乗則でスケールする (Kaplan *et al.*, 2020)。Transformer のスケーラビリティは、より大きなデータで学習されたより大規模な Vision Transformer による大幅な性能向上からも裏付けられている (Section 14.8 で議論)。より最近の成功例には Gato があり、これは Atari をプレイし、画像にキャプションを付け、会話し、ロボットとして行動できる汎用モデルである (Reed *et al.*, 2022)。Gato は、テキスト、画像、関節トルク、ボタン押下など多様なモダリティで事前学習されたときにうまくスケールする単一の Transformer である。特筆すべきことに、このようなマルチモーダルデータはすべて、平坦なトークン列として直列化され、Transformer によってテキストトークン (Section 14.7) や画像パッチ (Section 14.8) と同様に処理できる。

マルチモーダルデータに対する Transformer 事前学習の説得力ある成功に先立って、Transformer は大量のテキストで広く事前学習されていた。もともと機械翻訳のために提案された Transformer アーキテクチャは、Fig. 14.7.1 に示すように、入力系列を表現するためのエンコーダと、ターゲット系列を生成するためのデコーダから構成される。主に、Transformer は 3 つの異なるモードで使用できる: エンコーダのみ、エンコーダ–デコーダ、および デコーダのみである。この章の最後に、これら 3 つのモードを概観し、Transformer の事前学習におけるスケーラビリティを説明する。

14.9.1 エンコーダのみ

Transformer エンコーダのみを使用する場合、入力トークン列は同じ数の表現に変換され、それらはさらに出力（たとえば分類）へ射影できる。Transformer エンコーダは自己注意層から構成され、すべての入力トークンが互いに注意を向ける。たとえば、Fig. 14.8.1 に示した Vision Transformer はエンコーダのみであり、入力画像パッチの列を特別な“<cls>”トークンの表現へ変換する。この表現はすべての入力トークンに依存するため、さらに分類ラベルへ射影される。この設計は、テキストで事前学習された先行のエンコーダのみ Transformer、BERT

(Bidirectional Encoder Representations from Transformers) (Devlin *et al.*, 2018) に着想を得ている。

BERT の事前学習

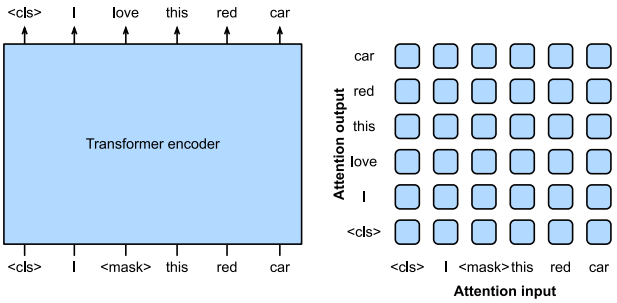


Fig. 14.9.1 左: マスク付き言語モデリングによる BERT の事前学習。マスクされた“love”トークンの予測は、“love”の前後にあるすべての入力トークンに依存する。右: Transformer エンコーダにおける注意パターン。縦軸上の各トークンは、横軸上のすべての入力トークンに注意を向ける。

BERT は マスク付き言語モデリングを用いてテキスト系列で事前学習される: ランダムにマスクされたトークンを含む入力テキストを Transformer エンコーダに与え、マスクされたトークンを予測する。Fig. 14.9.1 に示すように、元のテキスト系列“I”, “love”, “this”, “red”, “car”の先頭に“<cls>”トークンを付加し、“<mask>”トークンが“love”をランダムに置き換える。その後、マスクされたトークン“love”とその予測との間のクロスエントロピー損失を、事前学習中に最小化する。Transformer エンコーダの注意パターン (Fig. 14.9.1 の右側) には制約がないため、すべてのトークンが互いに注意を向けられることに注意されたい。したがって、“love”の予測は系列中の前後の入力トークンに依存する。これが BERT が「双方向エンコーダ」である理由である。手作業のラベル付けを必要とせず、書籍や Wikipedia から得られる大規模テキストデータを BERT の事前学習に利用できる。

BERT のファインチューニング

事前学習済み BERT は、単一テキストまたはテキスト対を含む下流のエンコーディングタスクに対して ファインチューニングできる。ファインチューニング中には、ランダムに初期化された追加層を BERT に加えることができ、これらのパラメータと事前学習済み BERT のパラメータの両方が、下流タスクの学習データに適合するように 更新される。

Fig. 14.9.2 は、感情分析のための BERT のファインチューニングを示している。Transformer エンコーダは事前学習済み BERT であり、テキスト系列を入力として受け取り、“<cls>”表現（入力全体のグローバル表現）を追加の全結合層に入力して感情を予測する。ファインチューニング中は、感情分析データにおける予測とラベルの間のクロスエントロピー損失を、勾配ベースのアルゴリズムで

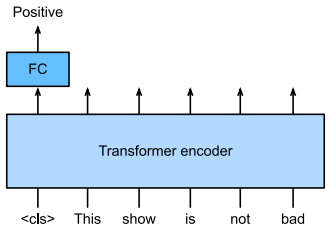


Fig. 14.9.2 感情分析のための BERT のファインチューニング。

最小化する。このとき、追加層はゼロから学習され、BERT の事前学習済みパラメータは更新される。BERT は感情分析だけにとどまらない。3.5 億パラメータの BERT が 2500 億トークンの学習トークンから学んだ汎用的な言語表現は、単一テキスト分類、テキスト対分類または回帰、テキストタグ付け、質問応答などの自然言語タスクにおいて最先端性能を押し上げた。

これらの下流タスクにはテキスト対理解が含まれることに気づくかもしれない。BERT の事前学習には、ある文が別の文に直後に続くかどうかを予測する別の損失がある。しかし、この損失は、同じサイズの BERT 変種を 2 兆トークンで事前学習した RoBERTa においては、後にあまり有用ではないことがわかった (Liu *et al.*, 2019)。BERT の他の派生モデルは、モデルアーキテクチャや事前学習目的を改善した。たとえば、ALBERT（パラメータ共有を強制）(Lan *et al.*, 2019)、SpanBERT（テキストのスパンを表現・予測）(Joshi *et al.*, 2020)、DistilBERT（知識蒸留による軽量化）(Sanh *et al.*, 2019)、ELECTRA（置換トークン検出）(Clark *et al.*, 2020) などである。さらに、BERT はコンピュータビジョンにおける Transformer 事前学習にも影響を与え、Vision Transformer (Dosovitskiy *et al.*, 2021)、Swin Transformer (Liu *et al.*, 2021)、MAE（masked autoencoders）(He *et al.*, 2022) などが生まれた。

14.9.2 エンコーダ–デコーダ

Transformer エンコーダは入力トークン列を同じ数の出力表現へ変換するため、エンコーダのみモードでは機械翻訳のように任意長の系列を生成できない。もともと機械翻訳のために提案されたように、Transformer アーキテクチャにはデコーダを組み込むことができ、デコーダはエンコーダ出力とデコーダ出力の両方を条件として、ターゲット系列をトークンごとに自己回帰的に予測し、任意長のターゲット系列を生成する: (i) エンコーダ出力を条件付けするために、エンコーダ–デコーダのクロスアテンション (Fig. 14.7.1 のデコーダのマルチヘッド注意) により、ターゲットトークンは すべての入力トークンに注意を向けられる; (ii) デコーダ出力を条件付けすることは、いわゆる 因果的注意（この名称は文献で一般的であるが、因果性の正しい研究とはほとんど関係がないため誤解を招く）パターン (Fig. 14.7.1 のデコーダのマスク付きマルチヘッド注意) によって実現され、任意のターゲットトークンはターゲット系列中の 過去と 現在のトークンにのみ注意を向けられる。

人手ラベル付きの機械翻訳データを超えてエンコーダ–デコーダ Transformer を

事前学習するために、BART (Lewis *et al.*, 2019) と T5 (Raffel *et al.*, 2020) は、大規模テキストコーパスで事前学習された 2 つの同時期に提案されたエンコーダ–デコーダ Transformer である。どちらも事前学習目的において元のテキストの復元を試みるが、前者は入力へのノイズ付加（たとえば、マスキング、削除、並べ替え、回転）を重視し、後者は包括的なアブレーション研究とともにマルチタスクの統一を強調している。

T5 の事前学習

事前学習済み Transformer エンコーダ–デコーダの例として、T5 (Text-to-Text Transfer Transformer) は多くのタスクを同じ text-to-text 問題として統一する: どのタスクでも、エンコーダの入力はタスク記述（たとえば“Summarize”, “:”）に続いてタスク入力（たとえば記事からのトークン列）となり、デコーダはタスク出力（たとえば入力記事を要約したトークン列）を予測する。text-to-text として機能するために、T5 は入力テキストに条件付けてあるターゲットテキストを生成するよう学習される。

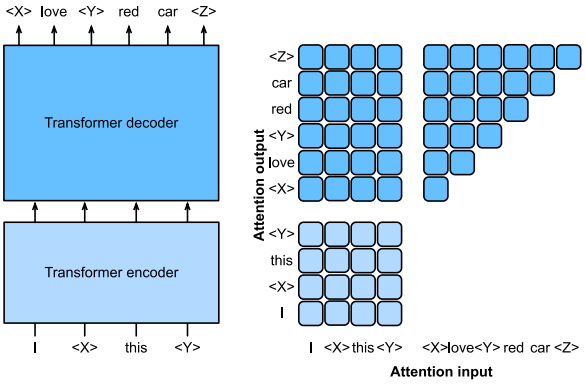


Fig. 14.9.3 左: 連続するスパンを予測することによる T5 の事前学習。元の文は“I”, “love”, “this”, “red”, “car”であり、“love”は特別な“<X>”トークンに置き換えられ、連続する“red”, “car”は特別な“<Y>”トークンに置き換えられる。ターゲット系列は特別な“<Z>”トークンで終わる。右: Transformer エンコーダ–デコーダにおける注意パターン。エンコーダの自己注意（下側の正方形）では、すべての入力トークンが互いに注意を向ける; エンコーダ–デコーダのクロスアテンション（上側の長方形）では、各ターゲットトークンがすべての入力トークンに注意を向ける; デコーダの自己注意（上側の三角形）では、各ターゲットトークンは現在および過去のターゲットトークンのみに注意を向ける（因果的）。

任意の元テキストから入力と出力を得るために、T5 は連続するスパンを予測するよう事前学習される。具体的には、テキスト中のトークンがランダムに特別なトークンで置き換えられ、各連続スパンは同じ特別トークンで置き換えられる。Fig. 14.9.3 の例を考えよう。元のテキストは“I”, “love”, “this”, “red”, “car”である。トークン“love”, “red”, “car”はランダムに特別なトークンで置き換えられる。“red”と“car”は連続するスパンなので、同じ特別トークンで置き換えられる。その結果、入力系列は“I”, “<X>”, “this”, “<Y>”となり、ターゲット系列は

“<X>”, “love”, “<Y>”, “red”, “car”, “<Z>”となる。ここで“<Z>”は終端を示す別の特別トークンである。Fig. 14.9.3 に示すように、デコーダは系列予測中に未来のトークンへ注意を向けないよう、因果的注意パターンを持つ。

T5 では、連続するスパンの予測は破損したテキストの復元とも呼ばれる。この目的のもと、T5 は C4 (Colossal Clean Crawled Corpus) データからの 1000 億トークンで事前学習された。このデータはウェブ上のクリーンな英語テキストから構成されている (Raffel *et al.*, 2020)。

T5 のファインチューニング

BERT と同様に、T5 もこのタスクを実行するために、タスク固有の学習データでファインチューニング（T5 パラメータの更新）する必要がある。BERT のファインチューニングとの主な違いは次のとおりである: (i) T5 の入力にはタスク記述が含まれる; (ii) T5 は Transformer デコーダにより任意長の系列を生成できる; (iii) 追加層は不要である。

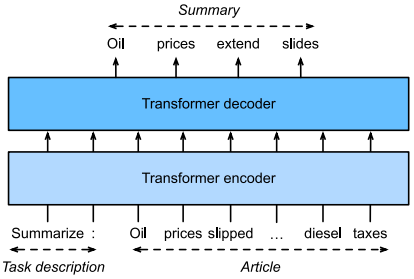


Fig. 14.9.4 テキスト要約のための T5 のファインチューニング。タスク記述と記事トークンの両方が Transformer エンコーダに入力され、要約を予測する。

Fig. 14.9.4 は、テキスト要約を例として T5 のファインチューニングを説明している。この下流タスクでは、タスク記述トークン“Summarize”, “:”に続いて記事トークンがエンコーダに入力される。

ファインチューニング後、110 億パラメータの T5（T5-11B）は、複数のエンコーディング（たとえば分類）および生成（たとえば要約）ベンチマークで最先端の結果を達成した。公開以来、T5 は後続研究で広く利用されている。たとえば、Switch Transformer は T5 を基に設計され、パラメータの一部のみを活性化して計算効率を高めている (Fedus *et al.*, 2022)。Imagen と呼ばれる text-to-image モデルでは、テキストは 46 億パラメータを持つ凍結された T5 エンコーダ（T5-XXL）に入力される (Saharia *et al.*, 2022)。Fig. 14.9.5 の写実的な text-to-image 例は、T5 エンコーダ単体でも、ファインチューニングなしでテキストを効果的に表現できる可能性を示唆している。

14.9.3 デコーダのみ

ここまでで、エンコーダのみおよびエンコーダ–デコーダ Transformer を概観した。これに対して、デコーダのみ Transformer は、Fig. 14.7.1 に示した元のエンコ



Fig. 14.9.5 Imagen モデルによる text-to-image の例。テキストエンコーダは T5 由来である（図は Saharia et al. (2022) より）。

ーダ–デコーダアーキテクチャから、エンコーダ全体と、エンコーダ–デコーダのクロスアテンションを含むデコーダのサブレイヤを取り除きる。今日では、デコーダのみ Transformer は、自己教師あり学習を通じて世界中に豊富に存在する未ラベルテキストコーパスを活用する大規模言語モデリング (Section 12.3) における 事実上のアーキテクチャとなっている。

GPT と GPT-2

学習目的として言語モデリングを用い、GPT (generative pre-training) モデルは Transformer デコーダをそのバックボーンとして選んだ (Radford *et al.*, 2018)。

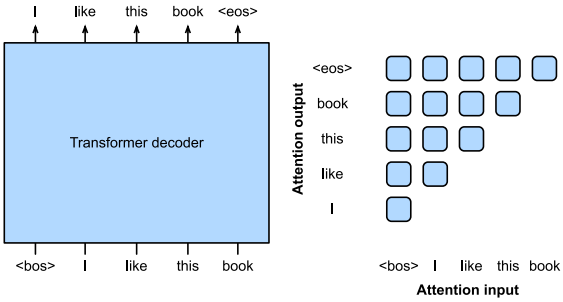


Fig. 14.9.6 左: 言語モデリングによる GPT の事前学習。ターゲット系列は入力系列を 1 トークンずらしたもの。“<bos>”と“<eos>”はそれぞれ系列の開始と終了を示す特別なトークンである。右: Transformer デコーダにおける注意パターン。縦軸上の各トークンは、横軸上の過去のトークンのみに注意を向ける（因果的）。

Section 12.3.3 で説明した自己回帰言語モデルの学習に従い、Fig. 14.9.6 は Transformer エンコーダを用いた GPT の事前学習を示しており、ターゲット系列は入力系列を 1 トークンずらしたものである。Transformer デコーダの注意パターンは、各トークンが過去のトークンにのみ注意を向けられることを強制することに注意されたい（未来のトークンはまだ選ばれていないため、注意を向けることができない）。

GPT は 1 億パラメータを持ち、個々の下流タスクごとにファインチューニングが必要である。はるかに大規模な Transformer デコーダ言語モデル GPT-2 は、1 年後に導入された (Radford *et al.*, 2019)。GPT の元の Transformer デコーダと比べて、事前正規化 (Section 14.8.3 で議論) と、改善された初期化および重みスケーリングが GPT-2 では採用された。400 GB のテキストで事前学習された 15 億パラメータの GPT-2 は、言語モデリングベンチマークで最先端の結果を得るとともに、パラメータやアーキテクチャを更新せずに複数の他タスクでも有望な結果を示した。

GPT-3 とその先

GPT-2 は、モデルを更新せずに同じ言語モデルを複数のタスクに使う可能性を示した。これは、勾配計算によるモデル更新を必要とするファインチューニングよりも、計算効率が高いである。

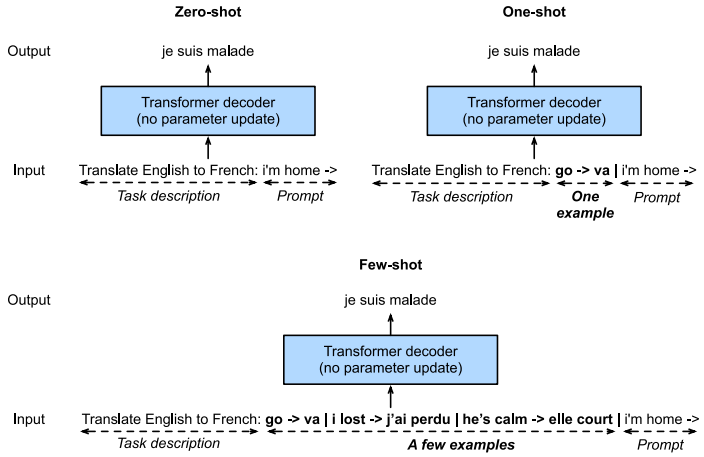


Fig. 14.9.7 言語モデル（Transformer デコーダ）を用いたゼロショット、ワンショット、フューショットの in-context learning。パラメータ更新は不要である。

パラメータ更新なしで言語モデルをより計算効率よく使う方法を説明する前に、Section 12.5 を思い出してほしい。そこでは、言語モデルはある接頭辞テキスト系列を条件としてテキスト系列を生成するよう学習できる。したがって、事前学習済み言語モデルは、タスク記述、タスク固有の入力–出力例、およびプロンプト（タスク入力）を含む入力系列を条件として、パラメータ更新なしでタスク出力を系列として生成できる。この学習パラダイムは *in-context learning* と呼ばれ (Brown *et al.*, 2020)、タスク固有の入力–出力例がない場合、1 つある場合、少数ある場合に応じて、ゼロショット、ワンショット、フューショットにさらに分類できる (Fig. 14.9.7)。

これら 3 つの設定は GPT-3 (Brown *et al.*, 2020) で検証され、その最大版は GPT-2 の約 2 桁大きいデータとモデルサイズを使用している。GPT-3 は GPT-2 の直接の後継である GPT-2 と同じ Transformer デコーダアーキテクチャを使用しているが、注意パターン (Fig. 14.9.6 の右側) は交互の層でより疎になっている。3000 億

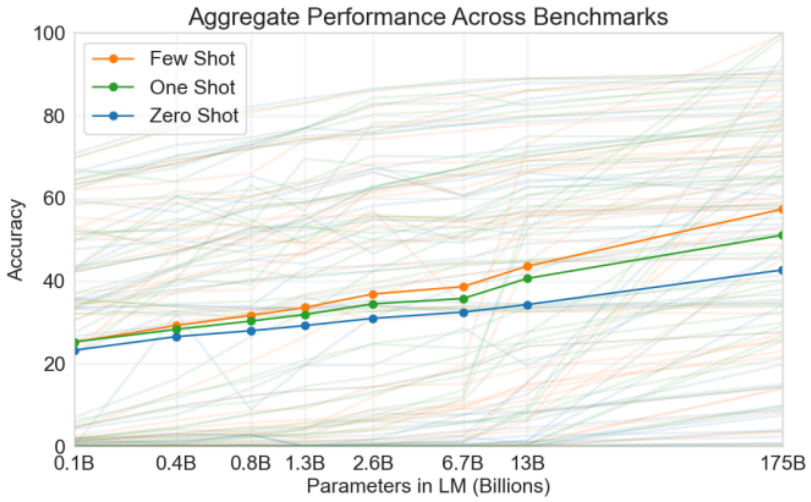


Fig. 14.9.8 GPT-3 の全 42 個の精度指標ベンチマークにおける集計性能（キャプションを調整し、図は Brown et al. (2020) より）。

トークンで事前学習された GPT-3 は、より大きなモデルサイズでより良い性能を示し、フューショット性能が最も急速に向上する (Fig. 14.9.8)。

その後の GPT-4 モデルは、報告書で技術的詳細を完全には公開しませんでした (OpenAI, 2023)。先行モデルとは対照的に、GPT-4 は大規模なマルチモーダルモデルであり、テキストと画像の両方を入力として受け取り、テキスト出力を生成できる。

14.9.4 スケーラビリティ

Fig. 14.9.8 は、GPT-3 言語モデルにおける Transformer のスケーラビリティを実証的に示している。言語モデリングに関しては、Transformer のスケーラビリティに関するより包括的な実証研究により、研究者たちは、より多くのデータと計算量でより大きな Transformer を学習することに可能性を見いだした (Kaplan *et al.*, 2020)。

Fig. 14.9.9 に示すように、べき乗則スケーリングは、モデルサイズ（埋め込み層を除くパラメータ数）、データセットサイズ（学習トークン数）、および学習計算量（PetaFLOP/s-days、埋め込み層を除く）に関して性能に観測される。一般に、これら 3 つの要因を同時に増やすと、より良い性能につながる。しかし、それらを どのように同時に増やすべきかは、依然として議論の的である (Hoffmann *et al.*, 2022)。

性能向上だけでなく、大規模モデルは小規模モデルよりもサンプル効率にも優れている。Fig. 14.9.10 は、大規模モデルが小規模モデルと同じ性能を達成するために必要な学習サンプル（処理トークン数）が少なく、性能が計算量に対して滑らかにスケールすることを示している。

(Kaplan *et al.*, 2020) における実証的スケーリング挙動は、その後の大規模 Trans-

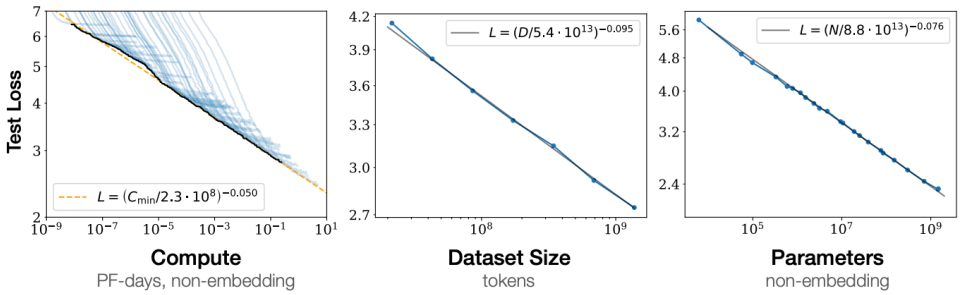


Fig. 14.9.9 Transformer 言語モデルの性能は、モデルサイズ、データセットサイズ、学習に使う計算量を増やすにつれて滑らかに向上する。最適性能を得るには、これら 3 つの要因を同時に拡大する必要がある。実証性能は、他の 2 つにボトルネックされていない場合、各要因それぞれに対してべき乗則の関係を持つ（キャプションを調整し、図は Kaplan et al. (2020) より）。

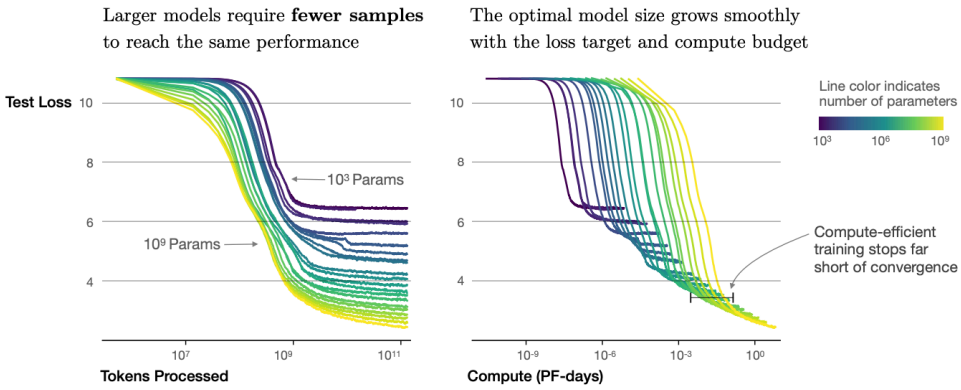


Fig. 14.9.10 Transformer 言語モデルの学習ラン（図は Kaplan et al. (2020) より）。

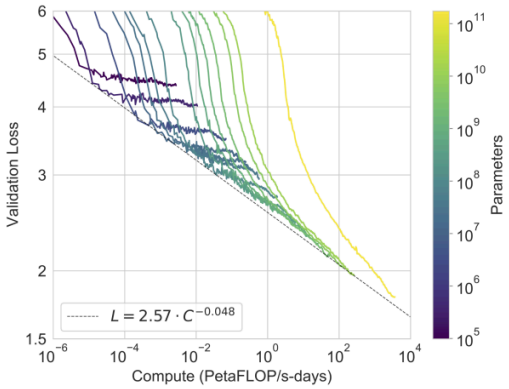


Fig. 14.9.11 GPT-3 の性能（クロスエントロピー検証損失）は、学習に使う計算量に対してべき乗則の傾向に従う。Kaplan et al. (2020) で観測されたべき乗則挙動は、予測曲線からの小さなずれのみで、さらに 2 桁の範囲にわたって続いている。埋め込みパラメータは計算量とパラメータ数から除外されている（キャプションを調整し、図は Brown et al. (2020) より）。

former モデルでも検証されている。たとえば、GPT-3 は Fig. 14.9.11 において、さらに 2 桁の範囲でこの仮説を支持した。

14.9.5 大規模言語モデル

GPT 系列における Transformer のスケーラビリティは、その後の大規模言語モデルに影響を与えた。GPT-2 の Transformer デコーダは、2700 億トークンで学習された 5300 億パラメータの Megatron-Turing NLG の学習に用いられた (Smith *et al.*, 2022)。GPT-2 の設計に従い、2800 億パラメータの Gopher (Rae *et al.*, 2021) は 3000 億トークンで事前学習され、多様なタスクで競争力のある性能を示した。同じアーキテクチャを継承し、Gopher と同じ計算予算を用いた Chinchilla (Hoffmann *et al.*, 2022) は、はるかに小さい（700 億パラメータ）モデルでありながら、より長く学習（1.4 兆トークン）し、多くのタスクで Gopher を上回り、パラメータ数よりもトークン数を重視する結果となった。言語モデリングのスケーリングの流れを継続するために、PaLM (Pathway Language Model) (Chowdhery *et al.*, 2022) は、修正された設計を持つ 5400 億パラメータの Transformer デコーダで、7800 億トークンで事前学習され、BIG-Bench ベンチマークにおいて平均的な人間性能を上回った (Srivastava *et al.*, 2022)。その後継である PaLM 2 (Anil *et al.*, 2023) は、データとモデルをおおむね 1:1 で拡大し、多言語能力と推論能力を向上させた。Minerva (Lewkowycz *et al.*, 2022) のように汎用モデル（PaLM）をさらに学習したものや、一般コーパスで学習されていない Galactica (Taylor *et al.*, 2022) などの他の大規模言語モデルも、有望な定量的推論能力および科学的推論能力を示した。

OPT (Open Pretrained Transformers) (Zhang *et al.*, 2022)、BLOOM (Scao *et al.*, 2022)、FALCON (Penedo *et al.*, 2023) のようなオープンソース公開は、大規模言語モデルの研究と利用を民主化した。推論時の計算効率に焦点を当てたオープンソースの Llama 1 (Touvron *et al.*, 2023a) は、通常よりも多くのトークンで学習することで、はるかに大きなモデルを上回った。更新版の Llama 2 (Touvron *et al.*, 2023b) は事前学習コーパスをさらに 40% 増やし、競合するクローズドソースモデルに匹敵しうる製品モデルにつながった。

Wei *et al.* (2022) は、大規模モデルには存在するが小規模モデルには存在しない、大規模言語モデルの創発的能力について議論した。しかし、単にモデルサイズを大きくするだけでは、モデルが人間の指示によりよく従うようになるわけではない。Sanh *et al.* (2021), Wei *et al.* (2021) は、指示によって記述されたさまざまなデータセットで大規模言語モデルをファインチューニングすると、保持されたタスクに対するゼロショット性能が向上することを見いだした。人間のフィードバックからの強化学習を用いて、Ouyang *et al.* (2022) は GPT-3 をファインチューニングし、多様な指示に従えるようにした。その結果として得られた InstructGPT に続き、ファインチューニングを通じて言語モデルを人間の意図に整合させる (Ouyang *et al.*, 2022) ChatGPT[83] は、人間のような応答（たとえばコードのデバッグや創作的な文章作成）を人間との会話に基づいて生成でき、多くの自然言語処理タスクをゼロショットで実行できる (Qin *et al.*, 2023)。Bai *et al.* (2022) は、指示チューニングの過程を部分的に自動化するために、人間の入力

83 

（たとえば人手ラベル付きデータ）をモデル出力で置き換えた。これは *AI* フィードバックからの強化学習とも呼ばれる。

大規模言語モデルは、プロンプトとも呼ばれる in-context learning を通じて、望ましいタスクをモデルに実行させるためのテキスト入力の与え方を定式化する、刺激的な可能性を提供する。特に、*chain-of-thought prompting* (Wei *et al.*, 2022) は、少数ショットの「質問、中間推論ステップ、答え」のデモンストレーションを用いる in-context learning 手法であり、大規模言語モデルの複雑な推論能力を引き出して、数学、常識、記号推論のタスクを解くのに役立つ。複数の推論経路をサンプリングすること (Wang *et al.*, 2023)、少数ショットのデモンストレーションを多様化すること (Zhang *et al.*, 2023)、複雑な問題を部分問題に分解すること (Zhou *et al.*, 2023) は、いずれも推論精度を改善できる。実際、各答えの直前に“Let’s think step by step”のような単純なプロンプトを与えるだけで、大規模言語モデルは十分な精度で ゼロショットの chain-of-thought 推論さえ実行できる (Kojima *et al.*, 2022)。テキストと画像の両方からなるマルチモーダル入力に対しても、言語モデルはテキスト入力のみを用いる場合より高い精度でマルチモーダル chain-of-thought 推論を実行できる (Zhang *et al.*, 2023)。

14.9.6 要約と考察

Transformer は、エンコーダのみ（たとえば BERT）、エンコーダ–デコーダ（たとえば T5）、およびデコーダのみ（たとえば GPT 系列）として事前学習されてきた。事前学習済みモデルは、モデル更新あり（たとえばファインチューニング）またはなし（たとえばフューショット）で、さまざまなタスクに適応できる。Transformer のスケーラビリティは、より大きなモデル、より多くの学習データ、より多くの学習計算量がより良い性能につながることを示唆している。Transformer はもともとテキストデータ向けに設計・事前学習されたため、この節はやや自然言語処理に寄っている。それでも、上で議論したこれらのモデルは、より最近の複数モダリティにまたがるモデルでもしばしば見られる。たとえば、(i) Chinchilla (Hoffmann *et al.*, 2022) はさらに Flamingo (Alayrac *et al.*, 2022) に拡張され、フューショット学習のための視覚言語モデルとなった; (ii) GPT-2 (Radford *et al.*, 2019) と Vision Transformer は、CLIP (Contrastive Language-Image Pre-training) (Radford *et al.*, 2021) においてテキストと画像をエンコードし、その画像埋め込みとテキスト埋め込みは後に DALL-E 2 の text-to-image システム (Ramesh *et al.*, 2022) に採用された。マルチモーダル事前学習における Transformer のスケーラビリティについてはまだ体系的研究がないが、Parti (Yu *et al.*, 2022) と呼ばれる全 Transformer の text-to-image モデルは、モダリティをまたいだスケーラビリティの可能性を示している: より大きな Parti は、より高忠実度な画像生成と、内容豊かなテキスト理解が可能です (Fig. 14.9.12)。

14.9.7 演習

1. 異なるタスクからなるミニバッチを用いて T5 をファインチューニングすることは可能か？なぜか、なぜではないか？GPT-2 ではどうだろうか。

2. 強力な言語モデルがあるとしたら、どのような応用が考えられるか？



Fig. 14.9.12 同じテキストから生成された、サイズが増加する Parti モデル（350M、750M、3B、20B）による画像例（例は Yu et al. (2022) より）。

3. 追加層を加えて言語モデルをテキスト分類にファインチューニングするよう求められたとする。どこにそれらを追加するか？なぜか？

4. ターゲット系列の予測全体を通して入力系列が常に利用可能な系列変換問題（たとえば機械翻訳）を考えよ。デコーダのみ Transformer でモデル化することの限界は何だろうか。なぜか？

15 最適化アルゴリズム

ここまで本書を順に読み進めてきたなら、深層学習モデルを訓練するために、すでにいくつかの最適化アルゴリズムを使ってきたはずである。それらは、モデルパラメータを継続的に更新し、訓練データセット上で評価した損失関数の値を最小化することを可能にしてくれた道具である。実際、最適化を単純な設定で目的関数を最小化するためのブラックボックス装置として扱うことに満足する人なら、「SGD」や「Adam」といった名前を持つ、そのような手続きの呪文のような一群が存在することを知っていれば十分だと思うかもしれない。

しかし、うまくやるには、もう少し深い知識が必要である。最適化アルゴリズムは深層学習において重要である。一方では、複雑な深層学習モデルの訓練には数時間、数日、あるいは数週間かかることがある。最適化アルゴリズムの性能は、モデルの訓練効率に直接影響する。他方では、さまざまな最適化アルゴリズムの原理と、それらのハイパーパラメータの役割を理解することで、ハイパーパラメータを目的に応じて調整し、深層学習モデルの性能を改善できるようになる。

この章では、深層学習でよく使われる最適化アルゴリズムを詳しく見ていく。深層学習で生じる最適化問題のほとんどは 非凸である。それでも、凸問題の文脈でのアルゴリズムの設計と解析は、非常に示唆に富むことが分かっている。そのため、この章では凸最適化の入門と、凸目的関数に対する非常に単純な確率的勾配降下法の証明も扱う。

15.1 最適化と深層学習

この節では、最適化と深層学習の関係、および深層学習における最適化の利用に伴う課題について議論する。深層学習の問題では、通常まず 損失関数を定義する。損失関数が得られれば、その損失を最小化しようとして最適化アルゴリズムを用いることができる。最適化では、損失関数はしばしば最適化問題の 目的関数と呼ばれる。慣例として、ほとんどの最適化アルゴリズムは 最小化を扱う。もし目的関数を最大化したい場合は、簡単な解決策がある。目的関数の符号を反転させればよい。

15.1.1 最適化の目標

最適化は深層学習における損失関数を最小化する手段を与えてくれるが、本質的には、最適化と深層学習の目標は根本的に異なる。前者は主として目的関数を最小化することに関心があるのに対し、後者は有限個のデータが与えられたときに適切なモデルを見つけることに関心がある。Section 6.6 では、この 2 つの目標の違いを詳しく議論した。たとえば、訓練誤差と汎化誤差は一般に異なる。最適化アルゴリズムの目的関数は通常、訓練データセットに基づく損失関数であるため、最適化の目標は訓練誤差を減らすことである。しかし、深層学習（より広くは統計的推論）の目標は汎化誤差を減らすことである。後者を達成するには、最適化アルゴリズムを用いて訓練誤差を減らすことに加えて、過学習にも注意を払う必要がある。

```
%matplotlib inline
from d2l import torch as d2l
import numpy as np
from mpl_toolkits import mplot3d
import torch
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()
```

```
def f(x):
    return x * d2l.cos(np.pi * x)

def g(x):
    return f(x) + 0.2 * d2l.cos(5 * np.pi * x)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import numpy as np
from mpl_toolkits import mplot3d
import tensorflow as tf
```

上述の異なる目標を説明するために、経験リスクとリスクを考えてみよう。Section 7.7.3 で述べたように、経験リスクは訓練データセット上の平均損失であり、一方リスクはデータ全体の母集団に対する期待損失である。以下では 2 つの関数を定義する。リスク関数 `f` と経験リスク関数 `g` である。訓練データが有限個しかないと仮定しよう。その結果、ここでの `g` は `f` よりも滑らかではない。

```
def f(x):
    return x * d2l.cos(np.pi * x)

def g(x):
    return f(x) + 0.2 * d2l.cos(5 * np.pi * x)
```

```
def f(x):
    return x * d2l.cos(np.pi * x)

def g(x):
    return f(x) + 0.2 * d2l.cos(5 * np.pi * x)
```

```
def annotate(text, xy, xytext):  #@save
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))

x = d2l.arange(0.5, 1.5, 0.01)
d2l.set_figsize((4.5, 2.5))
d2l.plot(x, [f(x), g(x)], 'x', 'risk')
annotate('min of\nempirical risk', (1.0, -1.2), (0.5, -1.1))
annotate('min of risk', (1.1, -1.05), (0.95, -0.5))
```

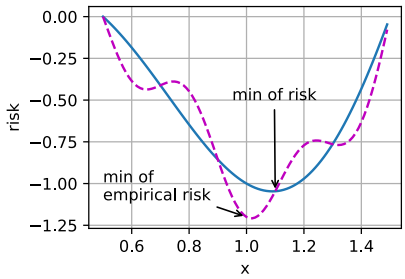

```
def f(x):
    return x * d2l.cos(np.pi * x)

def g(x):
    return f(x) + 0.2 * d2l.cos(5 * np.pi * x)
```


以下のグラフは、訓練データセット上の経験リスクの最小値が、リスク（汎化誤差）の最小値とは異なる位置にある場合があることを示している。


```
def annotate(text, xy, xytext):  #@save
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))

x = d2l.arange(0.5, 1.5, 0.01)
d2l.set_figsize((4.5, 2.5))
d2l.plot(x, [f(x), g(x)], 'x', 'risk')
annotate('min of\nempirical risk', (1.0, -1.2), (0.5, -1.1))
annotate('min of risk', (1.1, -1.05), (0.95, -0.5))
```

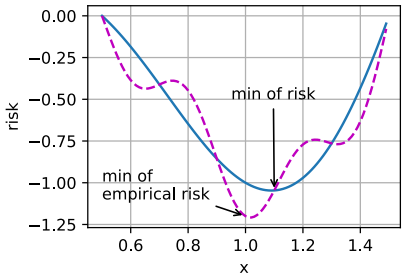

```
def annotate(text, xy, xytext):  #@save
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))
```


(continues on next page)

(continued from previous page)

```
x = d2l.arange(0.5, 1.5, 0.01)
d2l.set_figsize((4.5, 2.5))
d2l.plot(x, [f(x), g(x)], 'x', 'risk')
annotate('min of\nempirical risk', (1.0, -1.2), (0.5, -1.1))
annotate('min of risk', (1.1, -1.05), (0.95, -0.5))
```

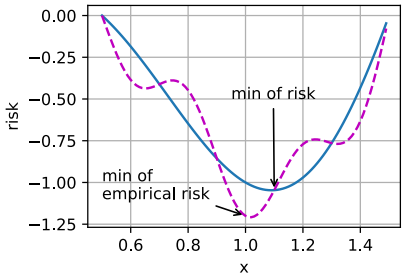
```
[07:06:10] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```




```
x = d2l.arange(-1.0, 2.0, 0.01)
d2l.plot(x, [f(x), ], 'x', 'f(x)')
annotate('local minimum', (-0.3, -0.25), (-0.77, -1.0))
annotate('global minimum', (1.1, -0.95), (0.6, 0.8))
```

```
def annotate(text, xy, xytext):  #@save
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))

x = d2l.arange(0.5, 1.5, 0.01)
d2l.set_figsize((4.5, 2.5))
d2l.plot(x, [f(x), g(x)], 'x', 'risk')
annotate('min of\nempirical risk', (1.0, -1.2), (0.5, -1.1))
annotate('min of risk', (1.1, -1.05), (0.95, -0.5))
```



15.1.2 深層学習における最適化の課題

この章では、モデルの汎化誤差ではなく、目的関数を最小化する際の最適化アルゴリズムの性能に特に焦点を当てる。Section 6.1 では、最適化問題における解析解と数値解を区別した。深層学習では、ほとんどの目的関数は複雑で、解析解を持たない。その代わりに、数値最適化アルゴリズムを使わなければならない。この章で扱う最適化アルゴリズムはすべて、このカテゴリに属する。

深層学習の最適化には多くの課題がある。特に厄介なのは、局所最小値、鞍点、そして勾配消失である。それらを見ていこう。

局所最小値

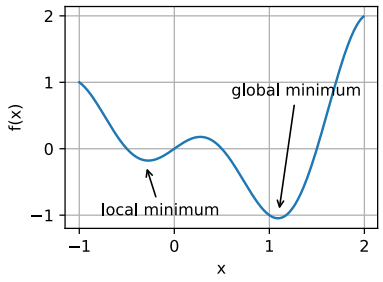
任意の目的関数 $f(x)$ について、ある点 x における $f(x)$ の値が、その近傍の他の任意の点における $f(x)$ の値よりも小さいなら、$f(x)$ は局所最小値である可能性がある。ある点 x における $f(x)$ の値が、定義域全体にわたる目的関数の最小値であるなら、$f(x)$ は大域最小値である。

たとえば、次の関数が与えられたとする。

$$f(x) = x \cdot \cos(\pi x) \text{ for } -1.0 \leq x \leq 2.0, \tag{15.1.1}$$

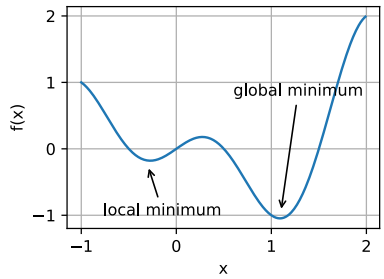
この関数の局所最小値と大域最小値を近似できる。

```
x = d2l.arange(-1.0, 2.0, 0.01)
d2l.plot(x, [f(x), ], 'x', 'f(x)')
annotate('local minimum', (-0.3, -0.25), (-0.77, -1.0))
annotate('global minimum', (1.1, -0.95), (0.6, 0.8))
```

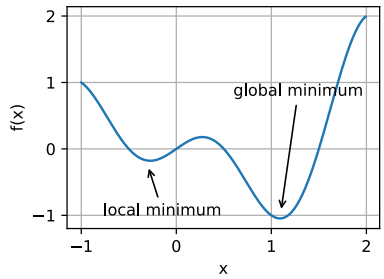



```
x = d2l.arange(-1.0, 2.0, 0.01)
d2l.plot(x, [f(x), ], 'x', 'f(x)')
annotate('local minimum', (-0.3, -0.25), (-0.77, -1.0))
annotate('global minimum', (1.1, -0.95), (0.6, 0.8))
```


```
x = d2l.arange(-2.0, 2.0, 0.01)
d2l.plot(x, [x**3], 'x', 'f(x)')
annotate('saddle point', (0, -0.2), (-0.52, -5.0))
```




```
x = d2l.arange(-1.0, 2.0, 0.01)
d2l.plot(x, [f(x), ], 'x', 'f(x)')
annotate('local minimum', (-0.3, -0.25), (-0.77, -1.0))
annotate('global minimum', (1.1, -0.95), (0.6, 0.8))
```

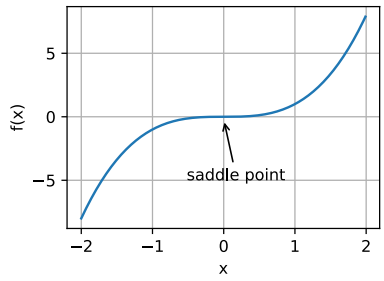


深層学習モデルの目的関数は通常、多くの局所最適解を持つ。最適化問題の数値解が局所最適解の近くにあるとき、最終反復で得られる数値解は、目的関数の勾配が 0 に近づくか 0 になるため、目的関数を 大域的にではなく 局所的にしか最小化しないかもしれない。ある程度のノイズがなければ、パラメータを局所最小値から押し出せないことがある。実際、これはミニバッチ確率的勾配降下法の有益な性質の 1 つであり、ミニバッチ間で勾配が自然に変動することで、パラメータを局所最小値から引き離すことができる。

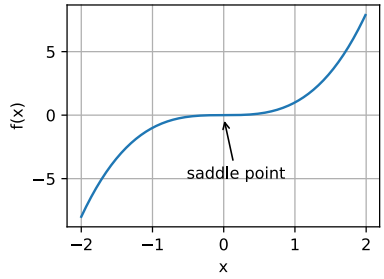
鞍点

局所最小値に加えて、鞍点も勾配が消失する別の原因である。鞍点とは、関数のすべての勾配が 0 になるものの、大域最小値でも局所最小値でもない位置のことである。関数 $f(x) = x^3$ を考えよう。その 1 階導関数と 2 階導関数は $x = 0$ で 0 になる。最適化は、この点で停滞する可能性があるが、ここは最小値ではない。

```
x = d2l.arange(-2.0, 2.0, 0.01)
d2l.plot(x, [x**3], 'x', 'f(x)')
annotate('saddle point', (0, -0.2), (-0.52, -5.0))
```

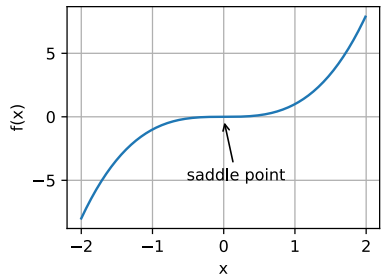



```
x = d2l.arange(-2.0, 2.0, 0.01)
d2l.plot(x, [x**3], 'x', 'f(x)')
annotate('saddle point', (0, -0.2), (-0.52, -5.0))
```




```
x = d2l.arange(-2.0, 5.0, 0.01)
d2l.plot(x, [d2l.tanh(x)], 'x', 'f(x)')
annotate('vanishing gradient', (4, 1), (2, 0.0))
```

```
x = d2l.arange(-2.0, 2.0, 0.01)
d2l.plot(x, [x**3], 'x', 'f(x)')
annotate('saddle point', (0, -0.2), (-0.52, -5.0))
```

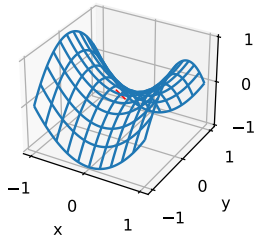


高次元では鞍点はさらに厄介である。以下の例を見てみよう。関数 $f(x, y) = x^2 - y^2$ を考える。この関数の鞍点は $(0, 0)$ にある。これは y に関しては最大値

であり、x に関しては最小値である。さらに、実際に 鞍のように見える。これが、この数学的性質の名前の由来である。

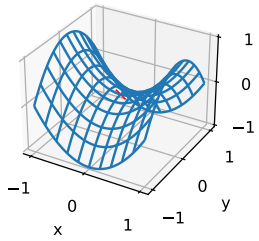
```
x, y = d2l.meshgrid(
    d2l.linspace(-1.0, 1.0, 101), d2l.linspace(-1.0, 1.0, 101))
z = x**2 - y**2

ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.plot([0], [0], [0], 'rx')
ticks = [-1, 0, 1]
d2l.plt.xticks(ticks)
d2l.plt.yticks(ticks)
ax.set_zticks(ticks)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y');
```



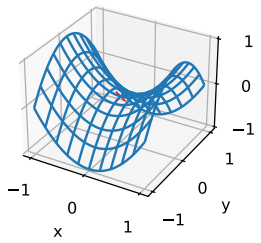

```
x, y = d2l.meshgrid(
    d2l.linspace(-1.0, 1.0, 101), d2l.linspace(-1.0, 1.0, 101))
z = x**2 - y**2

ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x.asnumpy(), y.asnumpy(), z.asnumpy(),
                  **{'rstride': 10, 'cstride': 10})
ax.plot([0], [0], [0], 'rx')
ticks = [-1, 0, 1]
d2l.plt.xticks(ticks)
d2l.plt.yticks(ticks)
ax.set_zticks(ticks)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y');
```



```
x, y = d2l.meshgrid(
    d2l.linspace(-1.0, 1.0, 101), d2l.linspace(-1.0, 1.0, 101))
z = x**2 - y**2

ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.plot([0], [0], [0], 'rx')
ticks = [-1, 0, 1]
d2l.plt.xticks(ticks)
d2l.plt.yticks(ticks)
ax.set_zticks(ticks)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y');
```



関数の入力が k 次元ベクトルで出力がスカラーであると仮定すると、そのヘッセ行列は k 個の固有値を持つ。関数の解は、関数の勾配が 0 である位置において、局所最小値、局所最大値、または鞍点になりえる。

- 関数のヘッセ行列の固有値が、勾配が 0 の位置で全て正なら、その関数は局所最小値を持つ。
- 関数のヘッセ行列の固有値が、勾配が 0 の位置で全て負なら、その関数は局所最大値を持つ。
- 関数のヘッセ行列の固有値が、勾配が 0 の位置で負と正の両方を含むなら、その関数は鞍点を持つ。

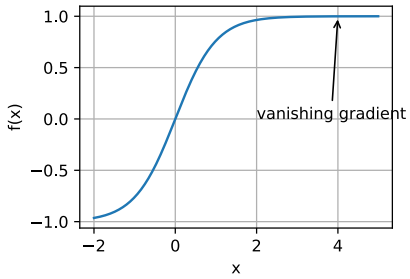
高次元問題では、少なくとも いくつかの固有値が負である確率はかなり高くなる。そのため、鞍点は局所最小値よりも起こりやすくなる。次の節で凸性を導入するときに、この状況に対するいくつかの例外を議論する。要するに、凸関数とはヘッセ行列の固有値が決して負にならない関数である。残念ながら、深層学習の問題の大半はこのカテゴリに入らない。それでも、最適化アルゴリズムを学ぶうえで非常に有用な道具である。

勾配消失

おそらく最も厄介な問題は勾配消失である。Section 8.1.2 で扱った、よく使われる活性化関数とその導関数を思い出されたい。たとえば、関数 $f(x) = \tanh(x)$ を最小化したいとして、たまたま $x = 4$ から始めたとする。見てのとおり、f の勾配はほとんど 0 である。より具体的には、$f'(x) = 1 - \tanh^2(x)$ なので、

$f'(4) = 0.0013$ である。その結果、進展が見られるまで最適化は長い間停滞する。これは、ReLU 活性化関数が導入される以前に深層学習モデルの訓練がかなり難しかった理由の 1 つである。

```
x = d2l.arange(-2.0, 5.0, 0.01)
d2l.plot(x, [d2l.tanh(x)], 'x', 'f(x)')
annotate('vanishing gradient', (4, 1), (2, 0.0))
```



見てきたように、深層学習の最適化には課題が山積している。幸いなことに、性能が良く、初心者でも使いやすい堅牢なアルゴリズムが幅広く存在する。さらに、必ずしも 唯一の最良解を見つける必要はない。局所最適解や、その近似解であっても、依然として非常に有用である。

15.1.3 まとめ

- 訓練誤差を最小化しても、汎化誤差を最小化する最良のパラメータ集合が見つかるとは限りません。
- 最適化問題には多くの局所最小値が存在しうる。
- 一般に問題は凸ではないため、さらに多くの鞍点が存在しうる。
- 勾配消失は最適化を停滞させることがある。しばしば、問題の再パラメータ化が役立つ。パラメータの良い初期化も有益である。

15.1.4 演習

1. たとえば隠れ層の次元が d で出力が 1 つの、単純な 1 隠れ層 MLP を考えよ。任意の局所最小値に対して、同一に振る舞う等価な解が少なくとも $d!$ 個あることを示せ。
2. 対称なランダム行列 $\mathbf{M}$ があり、その要素 $M_{ij} = M_{ji}$ はそれぞれある確率分布 p_{ij} から生成されると仮定する。さらに $p_{ij}(x) = p_{ij}(-x)$、すなわち分布が対称であると仮定する（詳細は例えば Wigner (1958) を参照）。
    1. 固有値の分布も対称であることを証明せよ。すなわち、任意の固有ベクトル $\mathbf{v}$ に対して、対応する固有値 λ が $P(\lambda > 0) = P(\lambda < 0)$ を満たす確率を示せ。

2. なぜ上の結果から *P(:raw-latex:`lambda `> 0) = 0.5* は導けないのか。

3. 深層学習の最適化に関わる他の課題として、どのようなものが考えられるか。

4. （実際の）ボールを（実際の）鞍の上で釣り合わせたいと仮定する。

   1. なぜこれは難しいのか。

   2. この効果を最適化アルゴリズムにも利用できるか。

15.2 凸性

凸性は最適化アルゴリズムの設計において重要な役割を果たす。
その大きな理由は、このような状況ではアルゴリズムの解析や検証がはるかに容易になるからである。
言い換えると、アルゴリズムが凸な設定でさえうまく動かないのであれば、通常、それ以外の状況で大きな成果を期待すべきではない。
さらに、深層学習における最適化問題は一般に非凸であるが、局所最小値の近傍では凸関数の性質の一部をしばしば示す。これにより、(Izmailov *et al.*, 2018) のような興味深い新しい最適化手法が生まれる。

```
%matplotlib inline
from d2l import torch as d2l
import numpy as np
from mpl_toolkits import mplot3d
import torch
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()
```

```
f = lambda x: 0.5 * x**2  # Convex
g = lambda x: d2l.cos(np.pi * x)  # Nonconvex
h = lambda x: d2l.exp(0.5 * x)  # Convex

x, segment = d2l.arange(-2, 2, 0.01), d2l.tensor([-1.5, 1])
d2l.use_svg_display()
_, axes = d2l.plt.subplots(1, 3, figsize=(9, 3))
for ax, func in zip(axes, [f, g, h]):
    d2l.plot([x, segment], [func(x), func(segment)], axes=ax)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import numpy as np
from mpl_toolkits import mplot3d
import tensorflow as tf
```

15.2.1 定義

凸解析に入る前に、凸集合と凸関数を定義する必要がある。これらは機械学習で広く用いられる数学的道具につながる。

凸集合

集合は凸性の基礎である。簡単に言えば、ベクトル空間内の集合 $\mathcal{X}$ は、任意の $a, b \in \mathcal{X}$ に対して、それらを結ぶ線分がやはり $\mathcal{X}$ に含まれるとき凸である。数学的には、すべての $\lambda \in [0, 1]$ について

$$\lambda a + (1 - \lambda) b \in \mathcal{X} \text{ whenever } a, b \in \mathcal{X}. \tag{15.2.1}$$

これは少し抽象的に聞こえるかもしれない。Fig. 15.2.1 を考えてみよう。最初の集合は、含まれない線分が存在するので凸ではない。他の 2 つの集合にはそのような問題はない。

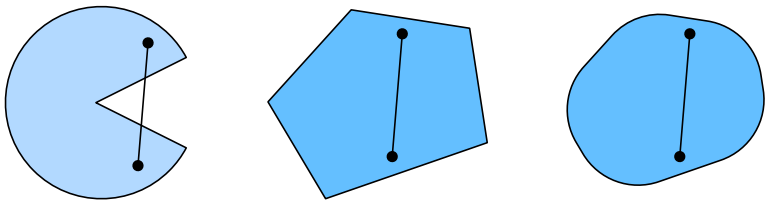


Fig. 15.2.1 最初の集合は非凸で、他の 2 つは凸である。

定義は、それを何かに使えない限り、単独ではあまり有用ではない。この場合、Fig. 15.2.2 に示すように交差を考えることができる。$\mathcal{X}$ と $\mathcal{Y}$ が凸集合だと仮定する。このとき $\mathcal{X} \cap \mathcal{Y}$ も凸である。これを見るには、任意の $a, b \in \mathcal{X} \cap \mathcal{Y}$ を考える。$\mathcal{X}$ と $\mathcal{Y}$ は凸なので、a と b を結ぶ線分は $\mathcal{X}$ と $\mathcal{Y}$ の両方に含まれる。したがって、それらは $\mathcal{X} \cap \mathcal{Y}$ にも含まれなければならず、これで定理が示される。

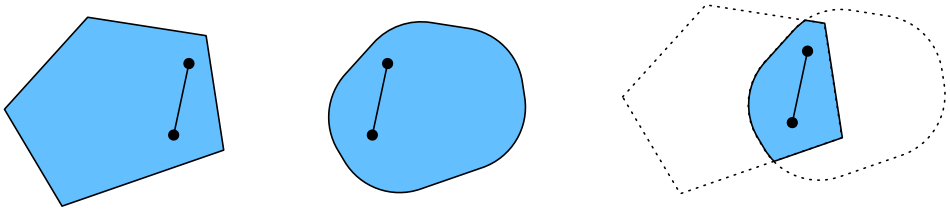


Fig. 15.2.2 2 つの凸集合の交差は凸である。

この結果はほとんど手間をかけずに強めることができる。すなわち、凸集合 $\mathcal{X}_i$ に対して、その交差 $\cap_i \mathcal{X}_i$ は凸である。逆が成り立たないことを見るには、互いに交わらない集合 $\mathcal{X} \cap \mathcal{Y} = \emptyset$ を考える。ここで $a \in \mathcal{X}$ と $b \in \mathcal{Y}$ を取る。Fig. 15.2.3 の a と b を結ぶ線分には、$\mathcal{X} \cap \mathcal{Y} = \emptyset$ を仮定しているので、$\mathcal{X}$ にも $\mathcal{Y}$ にも属さない部分が含まれていなければならない。したがって、その線分は $\mathcal{X} \cup \mathcal{Y}$ にも含まれず、一般に凸集合の和集合は凸である必要がないことが示される。

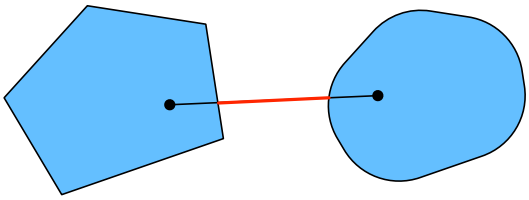


Fig. 15.2.3 2 つの凸集合の和集合は凸であるとは限らない。

通常、深層学習の問題は凸集合上で定義される。たとえば、$\mathbb{R}^d$、すなわち実数からなる d 次元ベクトルの集合は凸集合である（結局のところ、$\mathbb{R}^d$ 内の任意の 2 点を結ぶ線分は $\mathbb{R}^d$ にとどまる）。場合によっては、$\{\mathbf{x}|\mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\| \leq r\}$ で定義される半径 r の球のように、長さに上限のある変数を扱う。

凸関数

凸集合が定義できたので、凸関数 f を導入できる。凸集合 $\mathcal{X}$ が与えられたとき、関数 $f : \mathcal{X} \to \mathbb{R}$ は、すべての $x, x' \in \mathcal{X}$ とすべての $\lambda \in [0, 1]$ について

$$\lambda f(x) + (1 - \lambda) f(x') \geq f(\lambda x + (1 - \lambda) x'). \tag{15.2.2}$$

を満たすとき 凸である。

これを示すために、関数のグラフを描いて条件を満たすか確認してみよう。以下では、凸関数と非凸関数の両方をいくつか定義する。

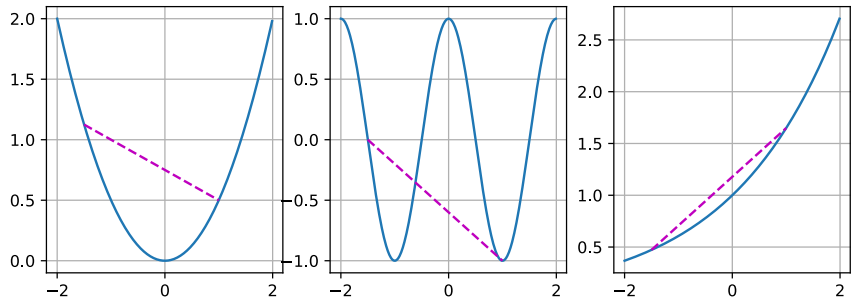
```
f = lambda x: 0.5 * x**2  # Convex
g = lambda x: d2l.cos(np.pi * x)  # Nonconvex
h = lambda x: d2l.exp(0.5 * x)  # Convex

x, segment = d2l.arange(-2, 2, 0.01), d2l.tensor([-1.5, 1])
d2l.use_svg_display()
_, axes = d2l.plt.subplots(1, 3, figsize=(9, 3))
for ax, func in zip(axes, [f, g, h]):
    d2l.plot([x, segment], [func(x), func(segment)], axes=ax)
```

```
f = lambda x: 0.5 * x**2  # Convex
g = lambda x: d2l.cos(np.pi * x)  # Nonconvex
h = lambda x: d2l.exp(0.5 * x)  # Convex

x, segment = d2l.arange(-2, 2, 0.01), d2l.tensor([-1.5, 1])
d2l.use_svg_display()
```

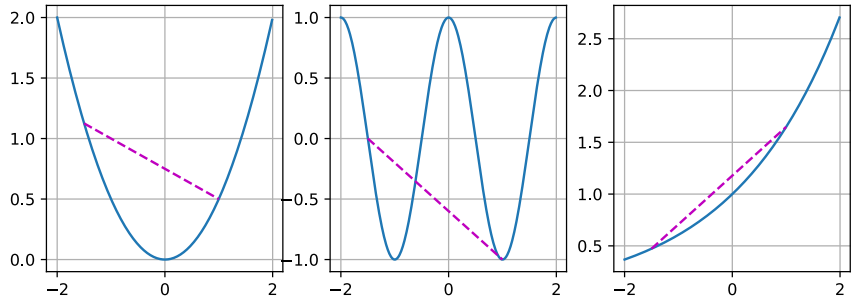
(continues on next page)



(continued from previous page)

```
_, axes = d2l.plt.subplots(1, 3, figsize=(9, 3))
for ax, func in zip(axes, [f, g, h]):
    d2l.plot([x, segment], [func(x), func(segment)], axes=ax)
```

```
[07:21:32] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

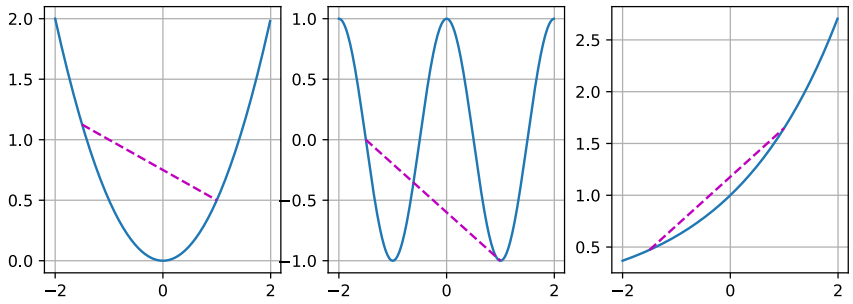


```
f = lambda x: (x - 1) ** 2
d2l.set_figsize()
d2l.plot([x, segment], [f(x), f(segment)], 'x', 'f(x)')
```

```
f = lambda x: 0.5 * x**2  # Convex
g = lambda x: d2l.cos(np.pi * x)  # Nonconvex
h = lambda x: d2l.exp(0.5 * x)  # Convex

x, segment = d2l.arange(-2, 2, 0.01), d2l.tensor([-1.5, 1])
d2l.use_svg_display()
_, axes = d2l.plt.subplots(1, 3, figsize=(9, 3))
for ax, func in zip(axes, [f, g, h]):
    d2l.plot([x, segment], [func(x), func(segment)], axes=ax)
```

予想どおり、余弦関数は 非凸であり、放物線と指数関数は凸である。条件が意味を持つためには、$\mathcal{X}$ が凸集合であることが必要である点に注意されたい。そうでなければ、$f(\lambda x + (1-\lambda)x')$ の値が定義できない可能性がある。



ジェンセンの不等式

凸関数 f が与えられたとき、最も有用な数学的道具の 1 つが ジェンセンの不等式である。これは凸性の定義の一般化に相当する。

$$\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right) \text{ and } E_X[f(X)] \geq f\left(E_X[X]\right), \tag{15.2.3}$$

ここで α_i は $\sum_i \alpha_i = 1$ を満たす非負の実数であり、X は確率変数である。言い換えると、凸関数の期待値は、その期待値に凸関数を適用したもの以上になる。後者のほうが通常はより簡単な式である。最初の不等式を証明するには、和の中の 1 項に対して順に凸性の定義を繰り返し適用する。

ジェンセンの不等式のよくある応用の 1 つは、より複雑な式をより簡単な式で上から抑えることである。たとえば、部分的に観測された確率変数の対数尤度に対して適用できる。つまり、次を用いる。

$$E_{Y \sim P(Y)}[-\log P(X \mid Y)] \geq -\log P(X), \tag{15.2.4}$$

なぜなら $\int P(Y)P(X \mid Y)dY = P(X)$ だからである。これは変分法で利用できる。ここで Y は通常、観測されていない確率変数であり、$P(Y)$ はその分布に関する最良の推定であり、$P(X)$ は Y を積分消去した分布である。たとえば、クラスタリングでは Y はクラスタラベルであり、$P(X \mid Y)$ はクラスタラベルを適用したときの生成モデルである。

15.2.2 性質

凸関数には多くの有用な性質がある。以下では、よく使われるものをいくつか説明する。

局所最小値は大域最小値

まず何よりも重要なのは、凸関数の局所最小値は大域最小値でもあることである。これは背理法で次のように証明できる。

凸集合 $\mathcal{X}$ 上で定義された凸関数 f を考える。$x^* \in \mathcal{X}$ が局所最小値だと仮定する。ある小さな正の値 p が存在して、$0 < |x - x^*| \leq p$ を満たす $x \in \mathcal{X}$ に対して $f(x^*) < f(x)$ が成り立つとする。

局所最小値 x^* が f の大域最小値ではないと仮定する。すると、$f(x') < f(x^*)$ を満たす $x' \in \mathcal{X}$ が存在する。また、$\lambda = 1 - \frac{p}{|x^* - x'|}$ のような $\lambda \in [0, 1)$ も存在し、$0 < |\lambda x^* + (1-\lambda)x' - x^*| \leq p$ となる。

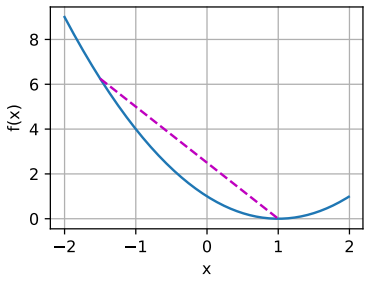
しかし、凸関数の定義によれば、次が成り立つ。

$$\begin{aligned} f(\lambda x^* + (1-\lambda)x') &\leq \lambda f(x^*) + (1-\lambda) f(x') \\ &< \lambda f(x^*) + (1-\lambda) f(x^*) \\ &= f(x^*), \end{aligned} \tag{15.2.5}$$

これは x^* が局所最小値であるという仮定に矛盾する。したがって、$f(x') < f(x^*)$ を満たす $x' \in \mathcal{X}$ は存在しない。局所最小値 x^* は大域最小値でもある。

たとえば、凸関数 $f(x) = (x-1)^2$ は $x = 1$ に局所最小値を持ち、それは同時に大域最小値でもある。

```
f = lambda x: (x - 1) ** 2
d2l.set_figsize()
d2l.plot([x, segment], [f(x), f(segment)], 'x', 'f(x)')
```



凸関数の局所最小値が大域最小値でもあるという事実は非常に便利である。これは、関数を最小化するときに「行き詰まる」ことがないことを意味する。ただし、これは大域最小値が複数存在しないという意味でも、そもそも存在しないという意味でもないことに注意されたい。たとえば、関数 $f(x) = \max(|x| - 1, 0)$ は区間 $[-1, 1]$ 上で最小値を取る。逆に、関数 $f(x) = \exp(x)$ は $\mathbb{R}$ 上で最小値を取らない。$x \to -\infty$ で 0 に漸近するが、$f(x) = 0$ となる x は存在しない。

凸関数の下位集合は凸

下位集合を用いると、凸集合を便利に定義できる。具体的には、凸集合 $\mathcal{X}$ 上で定義された凸関数 f に対して、任意の下位集合

$$\mathcal{S}_b \stackrel{\text{def}}{=} \{x | x \in \mathcal{X} \text{ and } f(x) \leq b\} \tag{15.2.6}$$

は凸である。

簡単に証明しよう。任意の $x, x' \in \mathcal{S}_b$ に対して、$\lambda \in [0, 1]$ のもとで $\lambda x + (1-\lambda)x' \in$

S_b を示せばよいことを想起されたい。$f(x) \le b$ かつ $f(x') \le b$ なので、凸性の定義より

$$f(\lambda x + (1-\lambda)x') \le \lambda f(x) + (1-\lambda) f(x') \le b. \tag{15.2.7}$$

凸性と 2 階微分

関数 $f : \mathbb{R}^n \to \mathbb{R}$ の 2 階微分が存在するなら、f が凸かどうかを調べるのは非常に簡単である。必要なのは、f のヘッセ行列が半正定値であるかどうかを確認することだけである。すなわち、$\nabla^2 f \succeq 0$、つまり、ヘッセ行列 $\nabla^2 f$ を $\mathbf{H}$ と書くと、すべての $\mathbf{x} \in \mathbb{R}^n$ について $\mathbf{x}^\top \mathbf{H} \mathbf{x} \ge 0$ が成り立つことである。たとえば、関数 $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}\|^2$ は、$\nabla^2 f = \mathbf{1}$、すなわちヘッセ行列が単位行列であるため凸である。

形式的には、2 回微分可能な 1 変数関数 $f : \mathbb{R} \to \mathbb{R}$ は、2 階導関数 $f'' \ge 0$ であることと同値に凸である。2 回微分可能な多変数関数 $f : \mathbb{R}^n \to \mathbb{R}$ については、ヘッセ行列 $\nabla^2 f \succeq 0$ であることと同値に凸である。

まず、1 変数の場合を証明する必要がある。f の凸性が $f'' \ge 0$ を意味することを見るには、次を用いる。

$$\frac{1}{2} f(x+\epsilon) + \frac{1}{2} f(x-\epsilon) \ge f\left(\frac{x+\epsilon}{2} + \frac{x-\epsilon}{2}\right) = f(x). \tag{15.2.8}$$

2 階導関数は有限差分の極限で与えられるので、次が従う。

$$f''(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) + f(x-\epsilon) - 2f(x)}{\epsilon^2} \ge 0. \tag{15.2.9}$$

$f'' \ge 0$ が f の凸性を意味することを見るには、$f'' \ge 0$ なら f' は単調非減少関数であることを用いる。$a < x < b$ を $\mathbb{R}$ 上の 3 点とし、$x = (1-\lambda)a + \lambda b$ かつ $\lambda \in (0,1)$ とする。平均値の定理によれば、$\alpha \in [a,x]$ と $\beta \in [x,b]$ が存在して

$$f'(\alpha) = \frac{f(x) - f(a)}{x - a} \text{ and } f'(\beta) = \frac{f(b) - f(x)}{b - x}. \tag{15.2.10}$$

単調性より $f'(\beta) \ge f'(\alpha)$ なので、

$$\frac{x-a}{b-a} f(b) + \frac{b-x}{b-a} f(a) \ge f(x). \tag{15.2.11}$$

$x = (1-\lambda)a + \lambda b$ なので、

$$\lambda f(b) + (1-\lambda) f(a) \ge f((1-\lambda)a + \lambda b), \tag{15.2.12}$$

したがって凸性が示される。

次に、多変数の場合を証明する前に補題が必要である。$f : \mathbb{R}^n \to \mathbb{R}$ が凸であることと、すべての $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ に対して

$$g(z) \stackrel{\text{def}}{=} f(z\mathbf{x} + (1-z)\mathbf{y}) \text{ where } z \in [0,1] \tag{15.2.13}$$

が凸であることは同値である。

f の凸性が g の凸性を意味することを証明するには、すべての $a, b, \lambda \in [0, 1]$ に対して（したがって $0 \leq \lambda a + (1 - \lambda)b \leq 1$）

$$\begin{aligned} & g(\lambda a + (1 - \lambda)b) \\ =& f\left((\lambda a + (1 - \lambda)b)\,\mathbf{x} + (1 - \lambda a - (1 - \lambda)b)\,\mathbf{y}\right) \\ =& f\left(\lambda\left(a\mathbf{x} + (1 - a)\mathbf{y}\right) + (1 - \lambda)\left(b\mathbf{x} + (1 - b)\mathbf{y}\right)\right) \\ \leq& \lambda f\left(a\mathbf{x} + (1 - a)\mathbf{y}\right) + (1 - \lambda) f\left(b\mathbf{x} + (1 - b)\mathbf{y}\right) \\ =& \lambda g(a) + (1 - \lambda) g(b). \end{aligned} \tag{15.2.14}$$

逆を証明するには、すべての $\lambda \in [0, 1]$ に対して

$$\begin{aligned} & f(\lambda \mathbf{x} + (1 - \lambda)\mathbf{y}) \\ =& g(\lambda \cdot 1 + (1 - \lambda) \cdot 0) \\ \leq& \lambda g(1) + (1 - \lambda) g(0) \\ =& \lambda f(\mathbf{x}) + (1 - \lambda) f(\mathbf{y}). \end{aligned} \tag{15.2.15}$$

最後に、上の補題と 1 変数の場合の結果を用いると、多変数の場合は次のように証明できる。多変数関数 $f : \mathbb{R}^n \to \mathbb{R}$ が凸であることと、すべての $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ に対して $g(z) \stackrel{\text{def}}{=} f(z\mathbf{x} + (1 - z)\mathbf{y})$、ただし $z \in [0, 1]$、が凸であることは同値である。1 変数の場合によれば、これは $g'' = (\mathbf{x} - \mathbf{y})^\top \mathbf{H}(\mathbf{x} - \mathbf{y}) \geq 0$（$\mathbf{H} \stackrel{\text{def}}{=} \nabla^2 f$）がすべての $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ について成り立つことと同値であり、これは半正定値行列の定義により $\mathbf{H} \succeq 0$ と同値である。

15.2.3 制約

凸最適化の優れた性質の 1 つは、制約を効率的に扱えることである。つまり、次の形の 制約付き最適化問題を解けることである。

$$\begin{aligned} & \underset{\mathbf{x}}{\text{minimize}}\ f(\mathbf{x}) \\ & \text{subject to } c_i(\mathbf{x}) \leq 0 \text{ for all } i \in \{1, \ldots, n\}, \end{aligned} \tag{15.2.16}$$

ここで f は目的関数、c_i は制約関数である。これが何を意味するかを見るために、$c_1(\mathbf{x}) = \|\mathbf{x}\|_2 - 1$ の場合を考える。このとき、パラメータ $\mathbf{x}$ は単位球に制約される。第 2 の制約が $c_2(\mathbf{x}) = \mathbf{v}^\top \mathbf{x} + b$ なら、これはすべての $\mathbf{x}$ が半空間上にあることに対応する。両方の制約を同時に満たすことは、球の一部を切り取ることに相当する。

ラグランジアン

一般に、制約付き最適化問題を解くのは難しい。その対処法の 1 つは物理学に由来し、かなり単純な直感に基づく。箱の中にボールがあると想像されたい。ボールは最も低い場所へ転がり、重力の力は箱の側面がボールに及ぼす力と釣り合う。要するに、目的関数の勾配（すなわち重力）は制約関数の勾配によって相殺される（ボールは壁が「押し返す」ことで箱の中にとどまる必要がある）。なお、いくつかの制約は有効でない場合がある。ボールに触れていない壁はボールに対して力を及ぼせない。

ラグランジアン L の導出は省略するが、上の考え方は次の鞍点最適化問題として表せる。

$$L(\mathbf{x}, \alpha_1, \ldots, \alpha_n) = f(\mathbf{x}) + \sum_{i=1}^{n} \alpha_i c_i(\mathbf{x}) \text{ where } \alpha_i \geq 0. \tag{15.2.17}$$

ここで変数 α_i（$i = 1, \ldots, n$）は、制約が適切に課されることを保証するいわゆる ラグランジュ乗数である。これらは、すべての i について $c_i(\mathbf{x}) \leq 0$ が成り立つのに十分な大きさに選ばれる。たとえば、自然に $c_i(\mathbf{x}) < 0$ となる任意の $\mathbf{x}$ に対しては、結局 $\alpha_i = 0$ を選ぶことになる。さらに、これは鞍点最適化問題であり、すべての α_i に関しては 最大化し、同時に $\mathbf{x}$ に関しては 最小化する。$L(\mathbf{x}, \alpha_1, \ldots, \alpha_n)$ という関数に至るまでの説明は豊富にある。ここでは、L の鞍点が元の制約付き最適化問題の最適解に対応することを知っていれば十分である。

ペナルティ

制約付き最適化問題を少なくとも 近似的に満たす 1 つの方法は、ラグランジアン L を修正することである。$c_i(\mathbf{x}) \leq 0$ を厳密に満たす代わりに、単に $\alpha_i c_i(\mathbf{x})$ を目的関数 $f(x)$ に加える。これにより、制約違反がひどくなりすぎることを防げる。

実際、私たちはずっとこの手法を使ってきた。Section 6.7 の重み減衰を考えてみよう。そこでは、$\mathbf{w}$ が大きくなりすぎないように、目的関数に $\frac{\lambda}{2}\|\mathbf{w}\|^2$ を加えている。制約付き最適化の観点からは、これはある半径 r に対して $\|\mathbf{w}\|^2 - r^2 \leq 0$ を保証するものと見なせる。λ の値を調整することで、$\mathbf{w}$ の大きさを変えられる。

一般に、ペナルティを加えることは、制約を近似的に満たすための良い方法である。実際には、これは厳密に満たす方法よりもはるかに頑健であることがわかる。さらに、非凸問題では、凸の場合に厳密な方法を魅力的にしていた多くの性質（たとえば最適性）はもはや成り立たない。

射影

制約を満たす別の戦略は射影である。これも以前に、たとえば Section 12.5 の勾配クリッピングで見た。そこでは、次のようにして勾配の長さを θ 以下に抑えた。

$$\mathbf{g} \leftarrow \mathbf{g} \cdot \min(1, \theta/\|\mathbf{g}\|). \tag{15.2.18}$$

これは $\mathbf{g}$ を半径 θ の球へ 射影したものである。より一般に、凸集合 $\mathcal{X}$ への射影は次で定義される。

$$\text{Proj}_{\mathcal{X}}(\mathbf{x}) = \underset{\mathbf{x}' \in \mathcal{X}}{\text{argmin}} \|\mathbf{x} - \mathbf{x}'\|, \tag{15.2.19}$$

これは $\mathbf{x}$ に最も近い $\mathcal{X}$ 内の点である。

射影の数学的定義は少し抽象的に聞こえるかもしれない。Fig. 15.2.4 はそれを

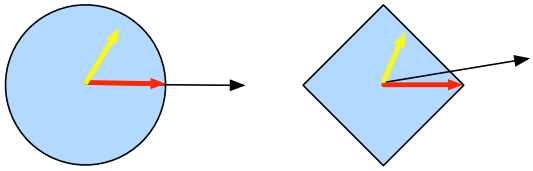


Fig. 15.2.4 凸射影。

もう少し明確に説明している。そこでは、円とひし形という 2 つの凸集合がある。両方の集合の内側にある点（黄色）は、射影の際に変化しない。両方の集合の外側にある点（黒）は、元の点（黒）に最も近い集合内の点（赤）へ射影される。ℓ_2 球では方向は変わらないが、ひし形の例からわかるように、一般にはそうとは限らない。

凸射影の用途の 1 つは、疎な重みベクトルを計算することである。この場合、重みベクトルを ℓ_1 球へ射影する。これは Fig. 15.2.4 のひし形の場合を一般化したものである。

15.2.4 まとめ

深層学習の文脈では、凸関数の主な目的は最適化アルゴリズムの動機づけを与え、それらを詳細に理解する助けとなることである。以下では、勾配降下法と確率的勾配降下法がどのように導かれるかを見ていく。

- 凸集合の交差は凸である。和集合はそうとは限らない。
- 凸関数の期待値は、その期待値に凸関数を適用したもの以上である（ジェンセンの不等式）。
- 2 回微分可能な関数は、そのヘッセ行列（2 階導関数の行列）が半正定値であることと同値に凸である。
- 凸制約はラグランジアンを通じて加えられる。実際には、目的関数にペナルティとして単純に加えてもよい。
- 射影は、元の点に最も近い凸集合内の点へ写す。

15.2.5 演習

1. 集合内の点どうしを結ぶすべての線を描き、それらの線が集合に含まれるかどうかを調べることで集合の凸性を確認したいとする。
   1. 境界上の点だけを調べれば十分であることを証明せよ。
   2. 集合の頂点だけを調べれば十分であることを証明せよ。
2. $\mathcal{B}_p[r] \stackrel{\text{def}}{=} \{\mathbf{x}|\mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\|_p \leq r\}$ を、p-ノルムを用いた半径 r の球とする。すべての $p \geq 1$ に対して $\mathcal{B}_p[r]$ が凸であることを証明せよ。
3. 凸関数 f と g が与えられたとき、$\max(f, g)$ も凸であることを示せ。$\min(f, g)$ が凸でないことを証明せよ。

4. softmax 関数の正規化が凸であることを証明せよ。より具体的には、$f(x) = \log \sum_i \exp(x_i)$ の凸性を証明せよ。
5. 線形部分空間、すなわち $\mathcal{X} = \{\mathbf{x}|\mathbf{W}\mathbf{x} = \mathbf{b}\}$ が凸集合であることを証明せよ。
6. $\mathbf{b} = \mathbf{0}$ の線形部分空間の場合、射影 $\text{Proj}_{\mathcal{X}}$ がある行列 $\mathbf{M}$ を用いて $\mathbf{M}\mathbf{x}$ と書けることを証明せよ。
7. 2 回微分可能な凸関数 f について、ある $\xi \in [0, \epsilon]$ が存在して $f(x + \epsilon) = f(x) + \epsilon f'(x) + \frac{1}{2}\epsilon^2 f''(x + \xi)$ と書けることを示せ。
8. 凸集合 $\mathcal{X}$ と 2 つのベクトル $\mathbf{x}, \mathbf{y}$ が与えられたとき、射影は距離を増やさない、すなわち $\|\mathbf{x} - \mathbf{y}\| \geq \|\text{Proj}_{\mathcal{X}}(\mathbf{x}) - \text{Proj}_{\mathcal{X}}(\mathbf{y})\|$ ことを証明せよ。

15.3 勾配降下法

この節では、勾配降下法の基礎となる概念を紹介する。深層学習で直接使われることはまれであるが、勾配降下法を理解することは確率的勾配降下法アルゴリズムを理解するうえで重要である。たとえば、学習率が大きすぎるために最適化問題が発散することがある。この現象は勾配降下法でもすでに観察できる。同様に、前処理は勾配降下法でよく使われる手法であり、より高度なアルゴリズムにも引き継がれる。まずは簡単な特殊ケースから始めよう。

15.3.1 1 次元の勾配降下法

1 次元の勾配降下法は、勾配降下アルゴリズムがなぜ目的関数の値を減少させうるのかを説明するのに最適な例である。ある連続微分可能な実数値関数 $f : \mathbb{R} \to \mathbb{R}$ を考える。テイラー展開を用いると

$$f(x + \epsilon) = f(x) + \epsilon f'(x) + O(\epsilon^2). \tag{15.3.1}$$

すなわち、1 次近似では $f(x + \epsilon)$ は点 x における関数値 $f(x)$ と 1 階導関数 $f'(x)$ によって与えられる。小さい ϵ に対して、負の勾配方向へ進めば f は減少すると考えるのは自然である。簡単のため、固定ステップサイズ $\eta > 0$ を選び、$\epsilon = -\eta f'(x)$ とする。これを上のテイラー展開に代入すると

$$f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + O(\eta^2 f'^2(x)). \tag{15.3.2}$$

導関数 $f'(x) \neq 0$ が消えないなら、$\eta f'^2(x) > 0$ なので進展がある。さらに、η を十分小さく選べば高次の項は無視できる。したがって

$$f(x - \eta f'(x)) \lessapprox f(x). \tag{15.3.3}$$

これは、x を反復更新する際に

$$x \leftarrow x - \eta f'(x) \tag{15.3.4}$$

を用いれば、関数 $f(x)$ の値が下がる可能性があることを意味する。したがって、勾配降下法ではまず初期値 x と定数 $\eta > 0$ を選び、それらを使って停止条件に達するまで x を継続的に反復更新する。たとえば、勾配の大きさ $|f'(x)|$ が十分小さくなったとき、あるいは反復回数がある値に達したときである。

簡単のため、目的関数 $f(x) = x^2$ を選んで、勾配降下法の実装方法を示す。$x = 0$ が $f(x)$ を最小化する解であることは分かっているが、それでもこの単純な関数を使って x がどのように変化するかを観察する。

```
%matplotlib inline
from d2l import torch as d2l
import numpy as np
import torch
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()
```

```
def f(x):  # 目的関数
    return x ** 2

def f_grad(x):  # 目的関数の勾配（導関数）
    return 2 * x
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import numpy as np
import tensorflow as tf
```

```
def f(x):  # 目的関数
    return x ** 2

def f_grad(x):  # 目的関数の勾配（導関数）
    return 2 * x
```

```
def f(x):  # 目的関数
    return x ** 2

def f_grad(x):  # 目的関数の勾配（導関数）
    return 2 * x
```

```
def gd(eta, f_grad):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x)
```

(continues on next page)

(continued from previous page)

```
        results.append(float(x))
    print(f'epoch 10, x: {x:f}')
    return results

results = gd(0.2, f_grad)
```

```
def f(x):  # 目的関数
    return x ** 2

def f_grad(x):  # 目的関数の勾配（導関数）
    return 2 * x
```

次に、初期値として $x = 10$ を用い、$\eta = 0.2$ と仮定する。勾配降下法で x を 10 回反復すると、最終的に x の値が最適解に近づくことが分かる。

```
def gd(eta, f_grad):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x)
        results.append(float(x))
    print(f'epoch 10, x: {x:f}')
    return results

results = gd(0.2, f_grad)
```

```
epoch 10, x: 0.060466
```

```
def gd(eta, f_grad):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x)
        results.append(float(x))
    print(f'epoch 10, x: {x:f}')
    return results

results = gd(0.2, f_grad)
```

```
epoch 10, x: 0.060466
```

```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = d2l.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results], [[f(x) for x in f_line], [
        f(x) for x in results]], 'x', 'f(x)', fmts=['-', '-o'])
```

(continues on next page)

(continued from previous page)

```
show_trace(results, f)
```

```
def gd(eta, f_grad):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x)
        results.append(float(x))
    print(f'epoch 10, x: {x:f}')
    return results

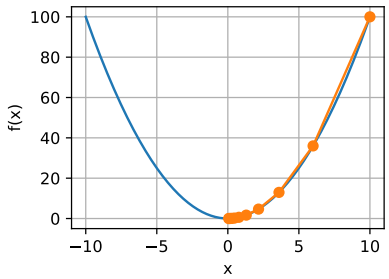
results = gd(0.2, f_grad)
```

```
epoch 10, x: 0.060466
```

x に関する最適化の進行は次のように描画できる。

```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = d2l.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results], [[f(x) for x in f_line], [
        f(x) for x in results]], 'x', 'f(x)', fmts=['-', '-o'])

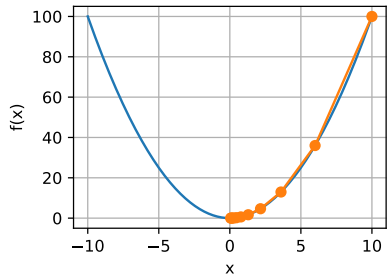
show_trace(results, f)
```




```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = d2l.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results], [[f(x) for x in f_line], [
        f(x) for x in results]], 'x', 'f(x)', fmts=['-', '-o'])

show_trace(results, f)
```

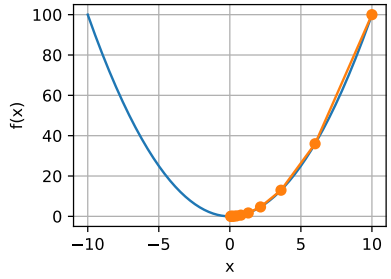
```
[07:06:17] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```




```
show_trace(gd(0.05, f_grad), f)
```

```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = d2l.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results], [[f(x) for x in f_line], [
        f(x) for x in results]], 'x', 'f(x)', fmts=['-', '-o'])

show_trace(results, f)
```

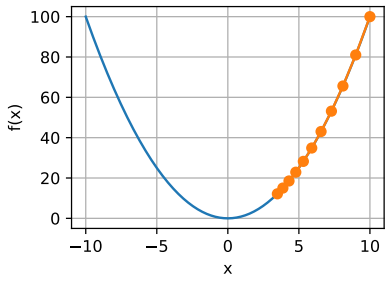


学習率

学習率 η はアルゴリズム設計者が設定できる。学習率が小さすぎると、x の更新が非常に遅くなり、よりよい解を得るために多くの反復が必要になる。このような場合に何が起こるかを見るため、同じ最適化問題で $\eta = 0.05$ の進行を考える。見て分かるように、10 ステップ後でもまだ最適解からかなり遠いままである。

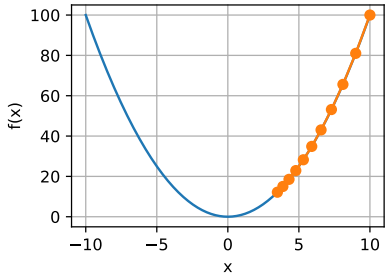
```
show_trace(gd(0.05, f_grad), f)
```

```
epoch 10, x: 3.486784
```




```
show_trace(gd(0.05, f_grad), f)
```

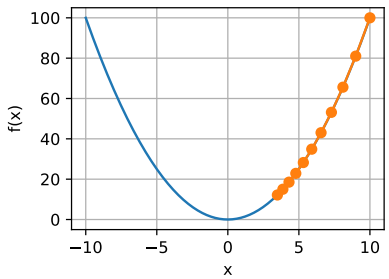
```
epoch 10, x: 3.486784
```




```
show_trace(gd(1.1, f_grad), f)
```

```
show_trace(gd(0.05, f_grad), f)
```

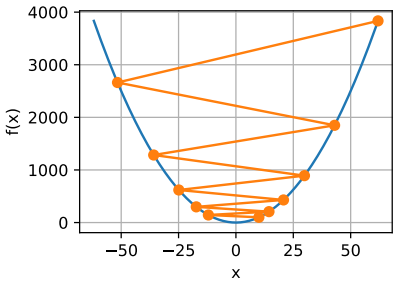
```
epoch 10, x: 3.486784
```



逆に、学習率が大きすぎると、$|\eta f'(x)|$ が 1 次テイラー展開の範囲を超えて大きくなりすぎることがある。つまり、(15.3.2) の項 $O(\eta^2 f'^2(x))$ が無視できなくなる可能性がある。この場合、x の反復が $f(x)$ の値を下げられるとは保証できない。たとえば、学習率を $\eta = 1.1$ にすると、x は最適解 $x = 0$ を飛び越えてしまい、徐々に発散する。

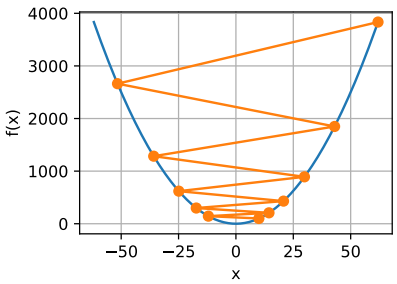
```
show_trace(gd(1.1, f_grad), f)
```

```
epoch 10, x: 61.917364
```




```
show_trace(gd(1.1, f_grad), f)
```

```
epoch 10, x: 61.917364
```




```
c = d2l.tensor(0.15 * np.pi)

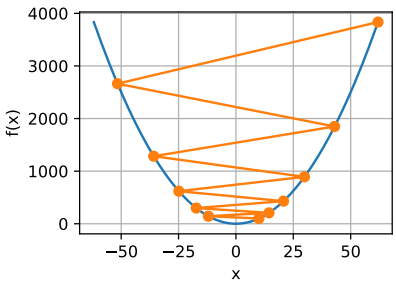
def f(x):  # 目的関数
    return x * d2l.cos(c * x)

def f_grad(x):  # 目的関数の勾配
    return d2l.cos(c * x) - c * x * d2l.sin(c * x)

show_trace(gd(2, f_grad), f)
```

```
show_trace(gd(1.1, f_grad), f)
```

```
epoch 10, x: 61.917364
```



局所最小値

非凸関数で何が起こるかを示すため、定数 c に対して $f(x) = x \cdot \cos(cx)$ を考える。この関数は無限個の局所最小値を持つ。学習率の選び方や問題の条件の良さによって、さまざまな解のうちのどれかに落ち着くことがある。以下の例は、（現実的ではない）高すぎる学習率が悪い局所最小値へ導くことを示している。

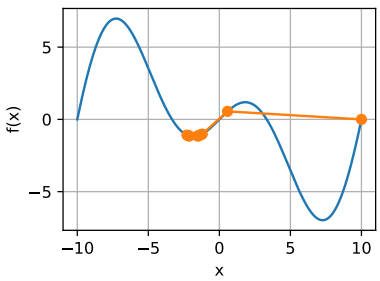
```
c = d2l.tensor(0.15 * np.pi)

def f(x):  # 目的関数
    return x * d2l.cos(c * x)

def f_grad(x):  # 目的関数の勾配
    return d2l.cos(c * x) - c * x * d2l.sin(c * x)

show_trace(gd(2, f_grad), f)
```

```
epoch 10, x: -1.528166
```



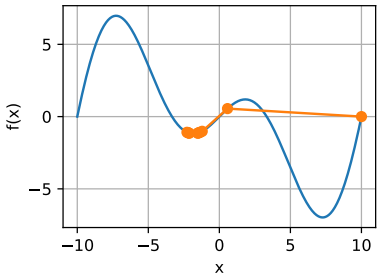
```
c = d2l.tensor(0.15 * np.pi)

def f(x):  # 目的関数
    return x * d2l.cos(c * x)

def f_grad(x):  # 目的関数の勾配
    return d2l.cos(c * x) - c * x * d2l.sin(c * x)

show_trace(gd(2, f_grad), f)
```

```
epoch 10, x: -1.528165
```




```
def train_2d(trainer, steps=20, f_grad=None):  #@save
    """ カスタマイズしたトレーナーで 2 次元目的関数を最適化する。"""
    # `s1` と `s2` は Momentum, adagrad, RMSProp で使われる内部状態変数
    x1, x2, s1, s2 = -5, -2, 0, 0
    results = [(x1, x2)]
    for i in range(steps):
        if f_grad:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2, f_grad)
        else:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
        results.append((x1, x2))
    print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results
```

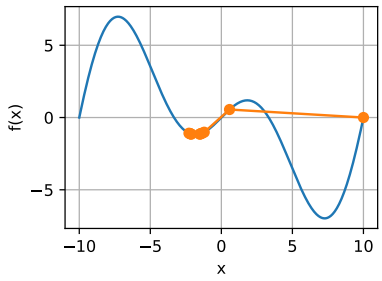
```
c = d2l.tensor(0.15 * np.pi)

def f(x):  # 目的関数
    return x * d2l.cos(c * x)

def f_grad(x):  # 目的関数の勾配
    return d2l.cos(c * x) - c * x * d2l.sin(c * x)

show_trace(gd(2, f_grad), f)
```

```
epoch 10, x: -1.528165
```



15.3.2 多変数の勾配降下法

1 変数の場合の直感がつかめたので、今度は $\mathbf{x} = [x_1, x_2, \ldots, x_d]^\top$ の場合を考えよう。つまり、目的関数 $f : \mathbb{R}^d \to \mathbb{R}$ はベクトルをスカラーに写像する。それに対応して、勾配も多変数になる。これは d 個の偏導関数からなるベクトルである。

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\right]^\top. \tag{15.3.5}$$

勾配の各偏導関数成分 $\partial f(\mathbf{x})/\partial x_i$ は、入力 x_i に関して点 $\mathbf{x}$ での f の変化率を表す。1 変数の場合と同様に、多変数関数に対する対応するテイラー近似を使えば、何をすべきかの手がかりが得られる。特に、

$$f(\mathbf{x} + \boldsymbol{\epsilon}) = f(\mathbf{x}) + \boldsymbol{\epsilon}^\top \nabla f(\mathbf{x}) + \mathcal{O}(\|\boldsymbol{\epsilon}\|^2). \tag{15.3.6}$$

言い換えると、$\boldsymbol{\epsilon}$ に関する 2 次の項までを見ると、最急降下方向は負の勾配 $-\nabla f(\mathbf{x})$ で与えられる。適切な学習率 $\eta > 0$ を選べば、典型的な勾配降下アルゴリズムが得られる。

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}). \tag{15.3.7}$$

実際にアルゴリズムがどう振る舞うかを見るため、入力として 2 次元ベクトル $\mathbf{x} = [x_1, x_2]^\top$ を取り、出力がスカラーである目的関数 $f(\mathbf{x}) = x_1^2 + 2x_2^2$ を構成しよう。勾配は $\nabla f(\mathbf{x}) = [2x_1, 4x_2]^\top$ である。初期位置 $[-5, -2]$ から勾配降下法で $\mathbf{x}$ がたどる軌跡を観察する。

まず、さらに 2 つの補助関数が必要である。1 つ目は更新関数を使って初期値に対して 20 回適用する。2 つ目は $\mathbf{x}$ の軌跡を可視化する。

```
def train_2d(trainer, steps=20, f_grad=None):  #@save
    """ カスタマイズしたトレーナーで 2 次元目的関数を最適化する。"""
    # `s1` と `s2` は Momentum, adagrad, RMSProp で使われる内部状態変数
    x1, x2, s1, s2 = -5, -2, 0, 0
    results = [(x1, x2)]
    for i in range(steps):
        if f_grad:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2, f_grad)
        else:
```

(continues on next page)

(continued from previous page)

```
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
        results.append((x1, x2))
    print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results
```

```
def train_2d(trainer, steps=20, f_grad=None):  #@save
    """ カスタマイズしたトレーナーで 2 次元目的関数を最適化する。"""
    # `s1` と `s2` は Momentum, adagrad, RMSProp で使われる内部状態変数
    x1, x2, s1, s2 = -5, -2, 0, 0
    results = [(x1, x2)]
    for i in range(steps):
        if f_grad:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2, f_grad)
        else:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
        results.append((x1, x2))
    print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results
```

```
def f_2d(x1, x2):  # 目的関数
    return x1 ** 2 + 2 * x2 ** 2

def f_2d_grad(x1, x2):  # 目的関数の勾配
    return (2 * x1, 4 * x2)

def gd_2d(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)

eta = 0.1
show_trace_2d(f_2d, train_2d(gd_2d, f_grad=f_2d_grad))
```

```
def train_2d(trainer, steps=20, f_grad=None):  #@save
    """ カスタマイズしたトレーナーで 2 次元目的関数を最適化する。"""
    # `s1` と `s2` は Momentum, adagrad, RMSProp で使われる内部状態変数
    x1, x2, s1, s2 = -5, -2, 0, 0
    results = [(x1, x2)]
    for i in range(steps):
        if f_grad:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2, f_grad)
        else:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
        results.append((x1, x2))
    print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results
```

```
def show_trace_2d(f, results):  #@save
    """ 最適化中の 2 次元変数の軌跡を表示する。"""
    d2l.set_figsize()
    d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')
```

(continues on next page)

(continued from previous page)

```
    x1, x2 = d2l.meshgrid(d2l.arange(-5.5, 1.0, 0.1),
                          d2l.arange(-3.0, 1.0, 0.1), indexing='ij')
    d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    d2l.plt.xlabel('x1')
    d2l.plt.ylabel('x2')
```

```
def show_trace_2d(f, results):  #@save
    """ 最適化中の 2 次元変数の軌跡を表示する。"""
    d2l.set_figsize()
    d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')
    x1, x2 = d2l.meshgrid(d2l.arange(-55, 1, 1),
                          d2l.arange(-30, 1, 1))
    x1, x2 = x1.asnumpy()*0.1, x2.asnumpy()*0.1
    d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    d2l.plt.xlabel('x1')
    d2l.plt.ylabel('x2')
```

```
c = d2l.tensor(0.5)

def f(x):  # 目的関数
    return d2l.cosh(c * x)

def f_grad(x):  # 目的関数の勾配
    return c * d2l.sinh(c * x)

def f_hess(x):  # 目的関数のヘッセ行列
    return c**2 * d2l.cosh(c * x)

def newton(eta=1):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x) / f_hess(x)
        results.append(float(x))
    print('epoch 10, x:', x)
    return results

show_trace(newton(), f)
```

```
def show_trace_2d(f, results):  #@save
    """ 最適化中の 2 次元変数の軌跡を表示する。"""
    d2l.set_figsize()
    d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')
    x1, x2 = d2l.meshgrid(d2l.arange(-5.5, 1.0, 0.1),
                          d2l.arange(-3.0, 1.0, 0.1))
    d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    d2l.plt.xlabel('x1')
    d2l.plt.ylabel('x2')
```

次に、学習率 $\eta = 0.1$ のときの最適化変数 $\mathbf{x}$ の軌跡を観察する。20 ステップ後に

は、$\mathbf{x}$ の値が最小値 $[0, 0]$ に近づいていることが分かる。進行はかなり安定しているが、やや遅めである。

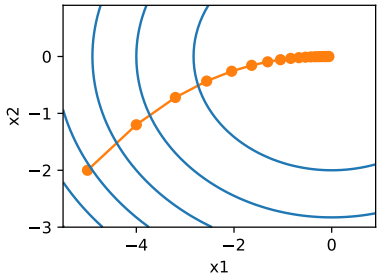
```
def f_2d(x1, x2):  # 目的関数
    return x1 ** 2 + 2 * x2 ** 2

def f_2d_grad(x1, x2):  # 目的関数の勾配
    return (2 * x1, 4 * x2)

def gd_2d(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)

eta = 0.1
show_trace_2d(f_2d, train_2d(gd_2d, f_grad=f_2d_grad))
```

```
epoch 20, x1: -0.057646, x2: -0.000073
```




```
def f_2d(x1, x2):  # 目的関数
    return x1 ** 2 + 2 * x2 ** 2

def f_2d_grad(x1, x2):  # 目的関数の勾配
    return (2 * x1, 4 * x2)

def gd_2d(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)

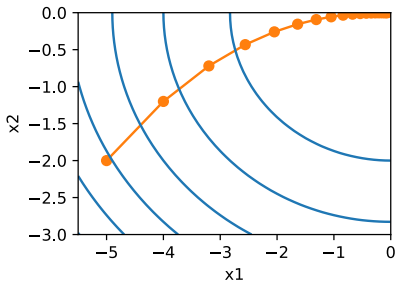
eta = 0.1
show_trace_2d(f_2d, train_2d(gd_2d, f_grad=f_2d_grad))
```

```
epoch 20, x1: -0.057646, x2: -0.000073
```

```
c = d2l.tensor(0.15 * np.pi)

def f(x):  # 目的関数
    return x * d2l.cos(c * x)
```

(continues on next page)



(continued from previous page)

```
def f_grad(x):  # 目的関数の勾配
    return d2l.cos(c * x) - c * x * d2l.sin(c * x)

def f_hess(x):  # 目的関数のヘッセ行列
    return - 2 * c * d2l.sin(c * x) - x * c**2 * d2l.cos(c * x)

show_trace(newton(), f)
```

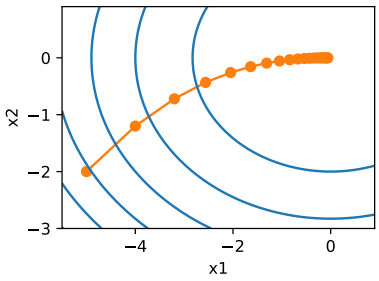
```
def f_2d(x1, x2):  # 目的関数
    return x1 ** 2 + 2 * x2 ** 2

def f_2d_grad(x1, x2):  # 目的関数の勾配
    return (2 * x1, 4 * x2)

def gd_2d(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)

eta = 0.1
show_trace_2d(f_2d, train_2d(gd_2d, f_grad=f_2d_grad))
```

```
epoch 20, x1: -0.057646, x2: -0.000073
```



15.3.3 適応的手法

Section 15.3.1 で見たように、学習率 η を「ちょうどよく」設定するのは難しい。小さすぎると進展がほとんどない。大きすぎると解が振動し、最悪の場合は発散する。学習率を自動的に決められたら、あるいはそもそも学習率を選ぶ必要がなくなったらどうだろうか。目的関数の値と勾配だけでなく、その曲率も見る 2 次の手法は、この場合に役立つ。これらの手法は計算コストのため深層学習に直接適用することはできないが、以下で述べる多くの望ましい性質を模倣した高度な最適化アルゴリズムを設計するうえで有用な直感を与えてくれる。

ニュートン法

ある関数 $f : \mathbb{R}^d \to \mathbb{R}$ のテイラー展開を見直すと、1 項目で止める必要はない。実際には

$$f(\mathbf{x} + \boldsymbol{\epsilon}) = f(\mathbf{x}) + \boldsymbol{\epsilon}^\top \nabla f(\mathbf{x}) + \frac{1}{2}\boldsymbol{\epsilon}^\top \nabla^2 f(\mathbf{x})\boldsymbol{\epsilon} + \mathcal{O}(\|\boldsymbol{\epsilon}\|^3). \tag{15.3.8}$$

煩雑な記法を避けるため、$\mathbf{H} \stackrel{\text{def}}{=} \nabla^2 f(\mathbf{x})$ を f のヘッセ行列と定義する。これは $d \times d$ 行列である。d が小さく単純な問題なら、$\mathbf{H}$ は容易に計算できる。一方、深層ニューラルネットワークでは、$\mathcal{O}(d^2)$ 個の要素を保存するコストのため、$\mathbf{H}$ は非常に大きくなりえる。さらに、逆伝播で計算するには高コストすぎる場合もある。ここではそうした点をいったん無視して、どのようなアルゴリズムが得られるかを見てみよう。

そもそも f の最小値では $\nabla f = 0$ が成り立つ。Section 5.4.3 の微積分の規則に従い、(15.3.8) を $\boldsymbol{\epsilon}$ について微分し、高次の項を無視すると

$$\nabla f(\mathbf{x}) + \mathbf{H}\boldsymbol{\epsilon} = 0 \text{ and hence } \boldsymbol{\epsilon} = -\mathbf{H}^{-1}\nabla f(\mathbf{x}). \tag{15.3.9}$$

つまり、最適化問題の一部としてヘッセ行列 $\mathbf{H}$ を逆行列化する必要がある。

簡単な例として、$f(x) = \frac{1}{2}x^2$ では $\nabla f(x) = x$ かつ $\mathbf{H} = 1$ である。したがって、どの x に対しても $\epsilon = -x$ が得られる。言い換えると、調整なしでたった *1* 回のステップで完全に収束できる。もっとも、これは少し幸運であった。$f(x + \epsilon) = \frac{1}{2}x^2 + \epsilon x + \frac{1}{2}\epsilon^2$ なので、テイラー展開が厳密に一致していたからである。

他の問題ではどうなるか見てみよう。定数 c に対して凸な双曲線余弦関数 $f(x) = \cosh(cx)$ を考えると、$x = 0$ にある大域的最小値に数回の反復で到達することが分かる。

```
c = d2l.tensor(0.5)

def f(x):  # 目的関数
    return d2l.cosh(c * x)

def f_grad(x):  # 目的関数の勾配
```

(continues on next page)

(continued from previous page)

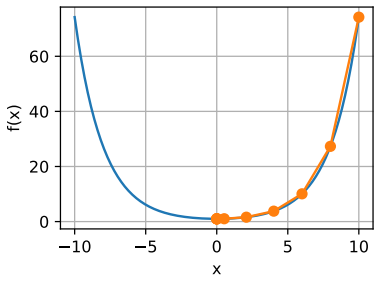
```
    return c * d2l.sinh(c * x)

def f_hess(x):  # 目的関数のヘッセ行列
    return c**2 * d2l.cosh(c * x)

def newton(eta=1):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x) / f_hess(x)
        results.append(float(x))
    print('epoch 10, x:', x)
    return results

show_trace(newton(), f)
```

```
epoch 10, x: tensor(0.)
```




```
c = d2l.tensor(0.5)

def f(x):  # 目的関数
    return d2l.cosh(c * x)

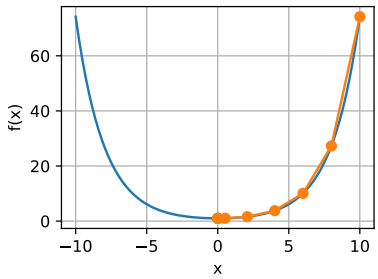
def f_grad(x):  # 目的関数の勾配
    return c * d2l.sinh(c * x)

def f_hess(x):  # 目的関数のヘッセ行列
    return c**2 * d2l.cosh(c * x)

def newton(eta=1):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x) / f_hess(x)
        results.append(float(x))
    print('epoch 10, x:', x)
    return results

show_trace(newton(), f)
```

```
epoch 10, x: 0.0
```




```
show_trace(newton(0.5), f)
```

```
c = d2l.tensor(0.5)

def f(x):  # 目的関数
    return d2l.cosh(c * x)

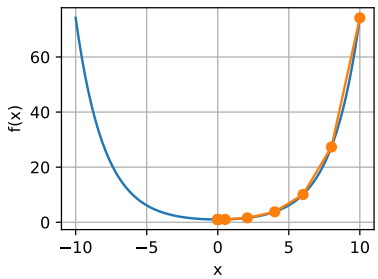
def f_grad(x):  # 目的関数の勾配
    return c * d2l.sinh(c * x)

def f_hess(x):  # 目的関数のヘッセ行列
    return c**2 * d2l.cosh(c * x)

def newton(eta=1):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x) / f_hess(x)
        results.append(float(x))
    print('epoch 10, x:', x)
    return results

show_trace(newton(), f)
```

```
epoch 10, x: tf.Tensor(0.0, shape=(), dtype=float32)
```



次に、定数 c に対して $f(x) = x\cos(cx)$ のような非凸関数を考えよう。ニュートン法ではヘッセ行列で割ることになる点に注意されたい。これは、2 階導関数が負であると、f の値を増加させる方向へ進んでしまう可能性があることを意味する。これはアルゴリズムの致命的な欠陥である。実際に何が起こるか見てみよう。

```
c = d2l.tensor(0.15 * np.pi)

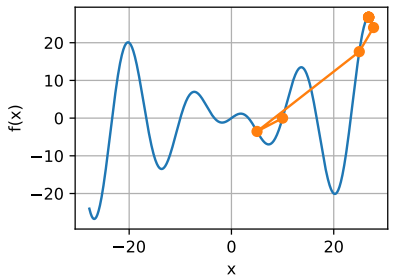
def f(x):  # 目的関数
    return x * d2l.cos(c * x)

def f_grad(x):  # 目的関数の勾配
    return d2l.cos(c * x) - c * x * d2l.sin(c * x)

def f_hess(x):  # 目的関数のヘッセ行列
    return - 2 * c * d2l.sin(c * x) - x * c**2 * d2l.cos(c * x)

show_trace(newton(), f)
```

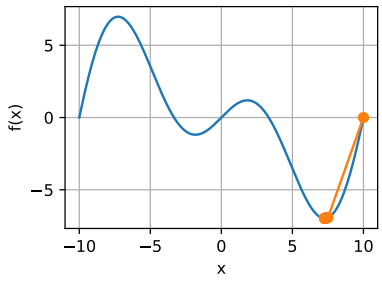
```
epoch 10, x: tensor(26.8341)
```



これはひどく失敗した。どうすれば修正できるだろうか。1 つの方法は、ヘッセ行列の絶対値を取って「修正」することである。別の戦略は、学習率を再導入することである。これは目的に反するように見えるかもしれないが、実はそうでもない。2 次情報があれば、曲率が大きいときには慎重になり、目的関数がより平坦なときには長いステップを取ることができる。少し小さめの学習率、たとえば $\eta = 0.5$ でこれがどう働くか見てみよう。かなり効率的なアルゴリズムになっていることが分かる。

```
show_trace(newton(0.5), f)
```

```
epoch 10, x: tensor(7.2699)
```



収束解析

ここでは、ある凸で 3 回微分可能な目的関数 f について、2 階導関数が 0 でない、すなわち $f'' > 0$ である場合のニュートン法の収束率のみを解析する。多変数の場合の証明は、以下の 1 次元の議論をそのまま拡張したものなので、直感の助けにならないため省略する。

$x^{(k)}$ を k^{th} 反復における x の値とし、$e^{(k)} \overset{\text{def}}{=} x^{(k)} - x^*$ を k^{th} 反復における最適解からの距離とする。テイラー展開により、条件 $f'(x^*) = 0$ は

$$0 = f'(x^{(k)} - e^{(k)}) = f'(x^{(k)}) - e^{(k)} f''(x^{(k)}) + \frac{1}{2}(e^{(k)})^2 f'''(\xi^{(k)}), \tag{15.3.10}$$

と書ける。ここで $\xi^{(k)} \in [x^{(k)} - e^{(k)}, x^{(k)}]$ となるある値である。上の展開を $f''(x^{(k)})$ で割ると

$$e^{(k)} - \frac{f'(x^{(k)})}{f''(x^{(k)})} = \frac{1}{2}(e^{(k)})^2 \frac{f'''(\xi^{(k)})}{f''(x^{(k)})}. \tag{15.3.11}$$

更新式 $x^{(k+1)} = x^{(k)} - f'(x^{(k)})/f''(x^{(k)})$ を思い出されたい。この更新式を代入し、両辺の絶対値を取ると

$$\left|e^{(k+1)}\right| = \frac{1}{2}(e^{(k)})^2 \frac{\left|f'''(\xi^{(k)})\right|}{f''(x^{(k)})}. \tag{15.3.12}$$

したがって、$\left|f'''(\xi^{(k)})\right| / (2f''(x^{(k)})) \leq c$ で抑えられる領域にいるなら、誤差は 2 乗的に減少し

$$\left|e^{(k+1)}\right| \leq c(e^{(k)})^2. \tag{15.3.13}$$

なお、最適化研究者はこれを線形収束と呼ぶが、$\left|e^{(k+1)}\right| \leq \alpha \left|e^{(k)}\right|$ のような条件は定数収束率と呼ばれる。この解析にはいくつかの注意点がある。第一に、急速収束の領域にいつ到達するかについては、実際にはあまり保証がない。分かるのは、その領域に入れば収束が非常に速いということだけである。第二に、この解析では f が高階導関数まで良い性質を持つことが必要である。要するに、f が値の変化の仕方に関して「予想外」の性質を持たないことを保証する必要がある。

前処理

完全なヘッセ行列を計算して保存するのが非常に高コストであることは、まったく驚くことではない。したがって、代替手段を見つけることが望ましい。改善の 1 つの方法が前処理である。これはヘッセ行列全体を計算する代わりに、対角成分だけを計算する。これにより、次の形の更新アルゴリズムが得られる。

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \mathrm{diag}(\mathbf{H})^{-1} \nabla f(\mathbf{x}). \tag{15.3.14}$$

これは完全なニュートン法ほど良くはないが、何もしないよりははるかに優れている。これが有効な理由を理解するために、一方の変数がミリメートル単位の高さを表し、もう一方がキロメートル単位の高さを表す状況を考えてみよう。どちらも自然なスケールはメートルだとすると、パラメータ化の不一致がひどいことになる。幸い、前処理を使えばこれを解消できる。実質的には、勾配降下法における前処理は、$\mathbf{x}$ の各変数（ベクトルの各座標）ごとに異なる学習率を選ぶことに相当する。後で見るように、前処理は確率的勾配降下法の最適化アルゴリズムにおけるいくつかの革新を支えている。

ラインサーチ付き勾配降下法

勾配降下法の主要な問題の 1 つは、目標を飛び越えてしまったり、進展が不十分だったりすることである。この問題の簡単な解決策は、勾配降下法と組み合わせてラインサーチを使うことである。つまり、$\nabla f(\mathbf{x})$ で与えられる方向を使い、その後、$f(\mathbf{x} - \eta \nabla f(\mathbf{x}))$ を最小化する学習率 η を二分探索で求める。

このアルゴリズムは高速に収束する（解析と証明については、たとえば Boyd and Vandenberghe (2004) を参照）。しかし、深層学習の目的ではこれはあまり現実的ではない。というのも、ラインサーチの各ステップで目的関数をデータセット全体に対して評価する必要があるからである。これはあまりにも高コストである。

15.3.4 まとめ

- 学習率は重要である。大きすぎると発散し、小さすぎると進展しない。
- 勾配降下法は局所最小値に捕まることがある。
- 高次元では学習率の調整は複雑である。
- 前処理はスケール調整に役立つ。
- ニュートン法は、凸問題で正しく機能し始めるとかなり高速である。
- 非凸問題に対して、何の調整もせずにニュートン法を使うのは危険である。

15.3.5 演習

1. 勾配降下法について、さまざまな学習率と目的関数を試してみよ。

2. 区間 $[a, b]$ における凸関数を最小化するためのラインサーチを実装せよ。
    1. 二分探索に導関数は必要か。つまり、$[a, (a+b)/2]$ と $[(a+b)/2, b]$ のどちらを選ぶかを決めるのに必要か。
    2. このアルゴリズムの収束率はどれくらい速いか。
    3. アルゴリズムを実装し、$\log(\exp(x) + \exp(-2x-3))$ の最小化に適用せよ。
3. 勾配降下法が非常に遅くなる $\mathbb{R}^2$ 上の目的関数を設計せよ。ヒント: 座標ごとに異なるスケールを使う。
4. 前処理を用いたニュートン法の軽量版を実装せよ。
    1. 対角ヘッセ行列を前処理として使え。
    2. 実際の（符号付きの）値ではなく、その絶対値を使え。
    3. 上の問題に適用せよ。
5. 上のアルゴリズムをいくつかの目的関数（凸でも非凸でもよい）に適用せよ。座標を 45 度回転させると何が起こるか。

15.4 確率的勾配降下法

前の章では、学習手順の中で確率的勾配降下法を使い続けてきたが、なぜそれがうまく働くのかについては説明していなかった。それを少し明らかにするために、Section 15.3 では勾配降下法の基本原理だけを説明した。この節では、確率的勾配降下法についてさらに詳しく議論する。

```
%matplotlib inline
from d2l import torch as d2l
import math
import torch
```

```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import np, npx
npx.set_np()
```

```
def f(x1, x2):  # Objective function
    return x1 ** 2 + 2 * x2 ** 2

def f_grad(x1, x2):  # Gradient of the objective function
    return 2 * x1, 4 * x2
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import math
import tensorflow as tf
```

15.4.1 確率的勾配更新

深層学習では、目的関数は通常、訓練データセット中の各サンプルに対する損失関数の平均である。n 個のサンプルからなる訓練データセットが与えられたとき、$\mathbf{x}$ をパラメータベクトルとして、インデックス i の訓練サンプルに関する損失関数を $f_i(\mathbf{x})$ とする。すると、目的関数は

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x}). \tag{15.4.1}$$

となる。

$\mathbf{x}$ における目的関数の勾配は次のように計算される。

$$\nabla f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{x}). \tag{15.4.2}$$

勾配降下法を用いると、各独立変数の反復ごとの計算コストは $O(n)$ であり、n に対して線形に増加する。したがって、訓練データセットが大きいほど、各反復における勾配降下法のコストは高くなる。

確率的勾配降下法（SGD）は、各反復の計算コストを削減する。確率的勾配降下法の各反復では、データサンプルのインデックス $i \in \{1, \ldots, n\}$ を一様にランダムサンプリングし、勾配 $\nabla f_i(\mathbf{x})$ を計算して $\mathbf{x}$ を更新する。

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}), \tag{15.4.3}$$

ここで η は学習率である。各反復の計算コストが、勾配降下法の $O(n)$ から定数の $O(1)$ に下がることがわかる。さらに、確率的勾配 $\nabla f_i(\mathbf{x})$ は全勾配 $\nabla f(\mathbf{x})$ の不偏推定量であることを強調しておく。なぜなら

$$\mathbb{E}_i \nabla f_i(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}). \tag{15.4.4}$$

だからである。

これは、平均的には確率的勾配が勾配のよい推定値であることを意味する。

ここでは、平均 0、分散 1 のランダムノイズを勾配に加えて確率的勾配降下法を模擬し、勾配降下法と比較してみる。

```
def f(x1, x2):  # Objective function
    return x1 ** 2 + 2 * x2 ** 2

def f_grad(x1, x2):  # Gradient of the objective function
    return 2 * x1, 4 * x2
```

```
def f(x1, x2):  # Objective function
    return x1 ** 2 + 2 * x2 ** 2

def f_grad(x1, x2):  # Gradient of the objective function
    return 2 * x1, 4 * x2
```

```
def constant_lr():
    return 1

eta = 0.1
lr = constant_lr  # Constant learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

```
def f(x1, x2):  # Objective function
    return x1 ** 2 + 2 * x2 ** 2

def f_grad(x1, x2):  # Gradient of the objective function
    return 2 * x1, 4 * x2
```

```
def sgd(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    # Simulate noisy gradient
    g1 += torch.normal(0.0, 1, (1,)).item()
    g2 += torch.normal(0.0, 1, (1,)).item()
    eta_t = eta * lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0)
```

```
def sgd(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    # Simulate noisy gradient
    g1 += d2l.normal(0.0, 1, (1,))
    g2 += d2l.normal(0.0, 1, (1,))
    eta_t = eta * lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0)
```

```
def exponential_lr():
    # Global variable that is defined outside this function and updated inside
    global t
    t += 1
    return math.exp(-0.1 * t)

t = 1
```

(continues on next page)

(continued from previous page)

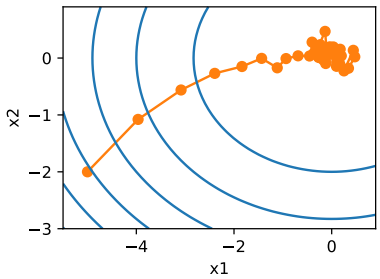
```
lr = exponential_lr
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=1000, f_grad=f_grad))
```

```
def sgd(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    # Simulate noisy gradient
    g1 += d2l.normal([1], 0.0, 1)
    g2 += d2l.normal([1], 0.0, 1)
    eta_t = eta * lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0)
```

```
def constant_lr():
    return 1

eta = 0.1
lr = constant_lr  # Constant learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

```
epoch 50, x1: -0.028379, x2: 0.194267
```



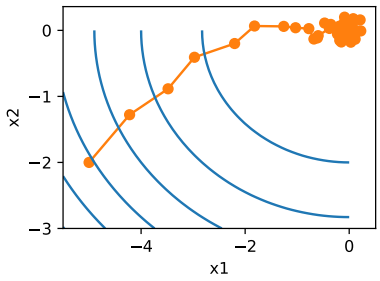

```
def constant_lr():
    return 1

eta = 0.1
lr = constant_lr  # Constant learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

```
[07:21:39] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
epoch 50, x1: -0.472513, x2: 0.110780
```

```
def polynomial_lr():
    # Global variable that is defined outside this function and updated inside
    global t
    t += 1
```

(continues on next page)



(continued from previous page)

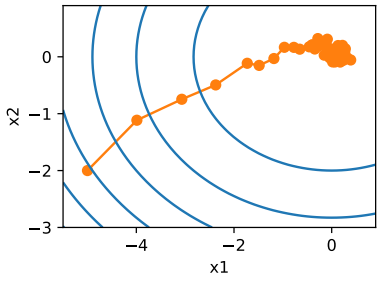
```
    return (1 + 0.1 * t) ** (-0.5)

t = 1
lr = polynomial_lr
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

```
def constant_lr():
    return 1

eta = 0.1
lr = constant_lr  # Constant learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

```
epoch 50, x1: 0.119305, x2: 0.031357
```



見てわかるように、確率的勾配降下法における変数の軌跡は、Section 15.3 で見た勾配降下法の軌跡よりもはるかにノイズが大きくなっている。これは勾配の確率的な性質によるものである。つまり、最小値付近に到達しても、$\eta \nabla f_i(\mathbf{x})$ によってその場で注入される不確実性の影響をなお受け続ける。50 ステップ後でも品質はまだあまり良くない。さらに悪いことに、ステップ数を増やしても改善しない（これを確認するために、より多くのステップ数で試してみることを勧める）。そこで残る唯一の選択肢は、学習率 η を変えることである。しかし、これを小さくしすぎると、最初のうちは意味のある進展が得られない。逆に大きすぎると、上で見たように良い解は得られない。こうした相反する目標

を解決する唯一の方法は、最適化が進むにつれて学習率を動的に減少させることである。

これが、sgd ステップ関数に学習率関数 lr を追加している理由でもある。上の例では、対応する lr 関数を定数に設定しているため、学習率スケジューリングの機能は使われていない。

15.4.2 動的な学習率

η を時間依存の学習率 $\eta(t)$ に置き換えると、最適化アルゴリズムの収束制御はより複雑になる。特に、η をどれくらい速く減衰させるべきかを決める必要がある。減衰が速すぎると、最適化を途中で打ち切ってしまう。遅すぎると、最適化に時間を浪費する。以下は、時間とともに η を調整するために使われる基本的な戦略のいくつかである（より高度な戦略は後で議論する）。

$$\begin{aligned} \eta(t) &= \eta_i \text{ if } t_i \leq t \leq t_{i+1} && \text{piecewise constant} \\ \eta(t) &= \eta_0 \cdot e^{-\lambda t} && \text{exponential decay} \\ \eta(t) &= \eta_0 \cdot (\beta t + 1)^{-\alpha} && \text{polynomial decay} \end{aligned} \tag{15.4.5}$$

最初の 区分定数の場合では、たとえば最適化の進展が停滞したときに学習率を下げる。これは深層ネットワークの学習でよく使われる戦略である。別の方法として、指数減衰によってもっと積極的に下げることもできる。残念ながら、これはしばしばアルゴリズムが収束する前に早期停止してしまう。よく使われる選択肢は、$\alpha = 0.5$ の 多項式減衰である。凸最適化の場合、この減衰率が良好に振る舞うことを示す証明が数多くある。

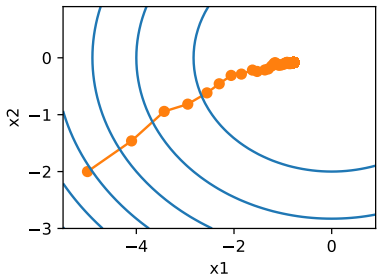
指数減衰が実際にはどのように見えるかを見てみよう。

```
def exponential_lr():
    # Global variable that is defined outside this function and updated inside
    global t
    t += 1
    return math.exp(-0.1 * t)

t = 1
lr = exponential_lr
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=1000, f_grad=f_grad))
```

```
epoch 1000, x1: -0.777940, x2: -0.083003
```

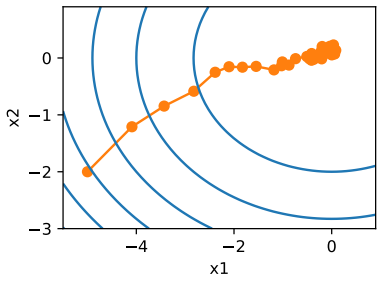
予想どおり、パラメータの分散は大幅に減少している。しかしその代償として、最適解 $\mathbf{x} = (0, 0)$ に収束できていない。1000 回の反復後でも、最適解からはまだ非常に遠いままである。実際、このアルゴリズムはまったく収束していない。一方、学習率がステップ数の平方根の逆数で減衰する多項式減衰を使うと、50 ステップ後には収束が改善する。




```
def polynomial_lr():
    # Global variable that is defined outside this function and updated inside
    global t
    t += 1
    return (1 + 0.1 * t) ** (-0.5)

t = 1
lr = polynomial_lr
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

```
epoch 50, x1: -0.007397, x2: 0.077371
```



学習率の設定方法には、さらに多くの選択肢がある。たとえば、小さな値から始めて、その後すばやく増やし、さらにゆっくりと再び減らすこともできる。あるいは、小さい学習率と大きい学習率を交互に使うことさえできる。このようなスケジュールには多様なものがある。ここでは、包括的な理論解析が可能な学習率スケジュール、すなわち凸設定における学習率に焦点を当てる。一般の非凸問題では、意味のある収束保証を得るのは非常に難しい。というのも、一般には非線形な非凸問題の最小化は NP 困難だからである。概説としては、Tibshirani 2015 の優れた lecture notes[84] を参照されたい。

84 

15.4.3 凸目的関数に対する収束解析

凸目的関数に対する確率的勾配降下法の以下の収束解析は任意選択であり、主として問題に対する直感を深めるためのものである。ここでは、最も簡単な証

明の一つ (Nesterov and Vial, 2000) に限定する。目的関数が特に良い性質を持つ場合など、はるかに高度な証明技法も存在する。

目的関数 $f(\boldsymbol{\xi}, \mathbf{x})$ が、すべての $\boldsymbol{\xi}$ に対して $\mathbf{x}$ に関して凸であると仮定する。より具体的には、次の確率的勾配降下法の更新を考える。

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta_t \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}), \tag{15.4.6}$$

ここで $f(\boldsymbol{\xi}_t, \mathbf{x})$ は、ステップ t においてある分布から引かれた訓練サンプル $\boldsymbol{\xi}_t$ に関する目的関数であり、$\mathbf{x}$ はモデルパラメータである。次で定義される

$$R(\mathbf{x}) = E_{\boldsymbol{\xi}}[f(\boldsymbol{\xi}, \mathbf{x})] \tag{15.4.7}$$

を期待リスク、その $\mathbf{x}$ に関する最小値を R^* とする。最後に、$\mathbf{x}^*$ を最小化解とする（$\mathbf{x}$ が定義される領域内に存在すると仮定する）。このとき、時刻 t における現在のパラメータ $\mathbf{x}_t$ とリスク最小化解 $\mathbf{x}^*$ との距離を追跡し、時間とともに改善するかどうかを調べることができる。

$$\begin{aligned} &\|\mathbf{x}_{t+1} - \mathbf{x}^*\|^2 \\ =&\|\mathbf{x}_t - \eta_t \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}) - \mathbf{x}^*\|^2 \\ =&\|\mathbf{x}_t - \mathbf{x}^*\|^2 + \eta_t^2 \|\partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x})\|^2 - 2\eta_t \left\langle \mathbf{x}_t - \mathbf{x}^*, \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}) \right\rangle. \end{aligned} \tag{15.4.8}$$

確率的勾配 $\partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x})$ の ℓ_2 ノルムがある定数 L によって抑えられると仮定すると、

$$\eta_t^2 \|\partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x})\|^2 \leq \eta_t^2 L^2. \tag{15.4.9}$$

ここで主に関心があるのは、$\mathbf{x}_t$ と $\mathbf{x}^*$ の距離が期待値の意味でどう変化するかである。実際、特定のステップ列に対しては、遭遇する $\boldsymbol{\xi}_t$ によって距離が増加することも十分ありえる。したがって、内積を上から抑える必要がある。任意の凸関数 f について $f(\mathbf{y}) \geq f(\mathbf{x}) + \langle f'(\mathbf{x}), \mathbf{y} - \mathbf{x} \rangle$ がすべての $\mathbf{x}$ と $\mathbf{y}$ に対して成り立つので、凸性より

$$f(\boldsymbol{\xi}_t, \mathbf{x}^*) \geq f(\boldsymbol{\xi}_t, \mathbf{x}_t) + \left\langle \mathbf{x}^* - \mathbf{x}_t, \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}_t) \right\rangle. \tag{15.4.10}$$

不等式 (15.4.9) と (15.4.10) の両方を (15.4.8) に代入すると、時刻 $t+1$ におけるパラメータ間の距離について次のような上界が得られる。

$$\|\mathbf{x}_t - \mathbf{x}^*\|^2 - \|\mathbf{x}_{t+1} - \mathbf{x}^*\|^2 \geq 2\eta_t (f(\boldsymbol{\xi}_t, \mathbf{x}_t) - f(\boldsymbol{\xi}_t, \mathbf{x}^*)) - \eta_t^2 L^2. \tag{15.4.11}$$

これは、現在の損失と最適損失の差が $\eta_t L^2/2$ を上回る限り、進展があることを意味する。この差はゼロに収束するはずなので、学習率 η_t も消失する必要がある。

次に、(15.4.11) の期待値を取る。すると

$$E\left[\|\mathbf{x}_t - \mathbf{x}^*\|^2\right] - E\left[\|\mathbf{x}_{t+1} - \mathbf{x}^*\|^2\right] \geq 2\eta_t [E[R(\mathbf{x}_t)] - R^*] - \eta_t^2 L^2. \tag{15.4.12}$$

最後のステップでは、$t \in \{1, \ldots, T\}$ に対する不等式を総和する。和は望遠和に

なり、下側の項を捨てると

$$\|\mathbf{x}_1 - \mathbf{x}^*\|^2 \geq 2\left(\sum_{t=1}^{T}\eta_t\right)\left[E[R(\mathbf{x}_t)] - R^*\right] - L^2\sum_{t=1}^{T}\eta_t^2. \tag{15.4.13}$$

$\mathbf{x}_1$ は与えられているので、期待値を外せることに注意されたい。最後に

$$\bar{\mathbf{x}} \stackrel{\text{def}}{=} \frac{\sum_{t=1}^{T}\eta_t\mathbf{x}_t}{\sum_{t=1}^{T}\eta_t}. \tag{15.4.14}$$

を定義する。

$$E\left(\frac{\sum_{t=1}^{T}\eta_t R(\mathbf{x}_t)}{\sum_{t=1}^{T}\eta_t}\right) = \frac{\sum_{t=1}^{T}\eta_t E[R(\mathbf{x}_t)]}{\sum_{t=1}^{T}\eta_t} = E[R(\mathbf{x}_t)], \tag{15.4.15}$$

であり、Jensen の不等式（(15.2.3) において $i = t, \alpha_i = \eta_t/\sum_{t=1}^{T}\eta_t$ と置く）と R の凸性より、$E[R(\mathbf{x}_t)] \geq E[R(\bar{\mathbf{x}})]$ が従うので、

$$\sum_{t=1}^{T}\eta_t E[R(\mathbf{x}_t)] \geq \sum_{t=1}^{T}\eta_t E\left[R(\bar{\mathbf{x}})\right]. \tag{15.4.16}$$

これを不等式 (15.4.13) に代入すると、次の上界が得られる。

$$[E[\bar{\mathbf{x}}]] - R^* \leq \frac{r^2 + L^2\sum_{t=1}^{T}\eta_t^2}{2\sum_{t=1}^{T}\eta_t}, \tag{15.4.17}$$

ここで $r^2 \stackrel{\text{def}}{=} \|\mathbf{x}_1 - \mathbf{x}^*\|^2$ は、初期パラメータの選択と最終結果との距離の上界である。要するに、収束速度は、確率的勾配のノルムがどのように抑えられているか（L）と、初期パラメータ値が最適値からどれだけ離れているか（r）に依存する。なお、この上界は $\mathbf{x}_T$ ではなく $\bar{\mathbf{x}}$ に関するものであることに注意されたい。これは、$\bar{\mathbf{x}}$ が最適化経路を平滑化したものだからである。r, L, T が既知であれば、学習率 $\eta = r/(L\sqrt{T})$ を選べる。すると上界は $rL/\sqrt{T}$ になる。つまり、最適解へ $O(1/\sqrt{T})$ の速度で収束する。

15.4.4 確率的勾配と有限サンプル

ここまで、確率的勾配降下法について少し大ざっぱに扱ってきた。通常は分布 $p(x, y)$ から、ラベル y_i を伴うインスタンス x_i をサンプリングし、それを何らかの形でモデルパラメータの更新に使うと仮定してきた。特に有限サンプルサイズの場合には、離散分布 $p(x, y) = \frac{1}{n}\sum_{i=1}^{n}\delta_{x_i}(x)\delta_{y_i}(y)$ が、ある関数 δ_{x_i} と δ_{y_i} に対して確率的勾配降下法を実行することを可能にすると単純に論じてきた。

しかし、実際にはそうしていたわけではない。この節の玩具例では、もともと確率的でない勾配にノイズを加えていただけであり、つまり (x_i, y_i) の組があるかのように振る舞っていた。ここではそれが正当化できることがわかる（詳細な議論は演習を参照されたい）。より問題なのは、これまでの議論では明らかにそうしていなかったことである。代わりに、すべてのインスタンスをちょうど一度ずつ走査していた。なぜこれが望ましいのかを見るために、逆に、離散

分布から n 個の観測を復元抽出する場合を考えよう。要素 i をランダムに選ぶ確率は $1/n$ である。したがって、それを少なくとも一度選ぶ確率は

$$P(\text{choose } i) = 1 - P(\text{omit } i) = 1 - (1 - 1/n)^n \approx 1 - e^{-1} \approx 0.63. \tag{15.4.18}$$

同様の考え方から、あるサンプル（すなわち訓練例）をちょうど一度選ぶ確率は

$$\binom{n}{1}\frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1} = \frac{n}{n-1}\left(1 - \frac{1}{n}\right)^n \approx e^{-1} \approx 0.37. \tag{15.4.19}$$

復元抽出は、非復元抽出と比べて分散を増やし、データ効率を下げる。したがって、実際には後者を行う（そしてこれが本書全体でのデフォルトの選択である）。最後に、訓練データセットを繰り返し走査するときには、毎回異なるランダム順序で走査されることに注意されたい。

15.4.5 まとめ

- 凸問題では、広い範囲の学習率に対して、確率的勾配降下法が最適解に収束することを証明できる。
- 深層学習では、一般にはそうではない。しかし、凸問題の解析は、最適化にどう取り組むべきかについて有用な洞察を与えてくれる。すなわち、学習率を徐々に下げること、ただし下げすぎないことである。
- 学習率が小さすぎても大きすぎても問題が起こる。実際には、適切な学習率は複数回の実験を経て初めて見つかることが多い。
- 訓練データセットのサンプル数が多いほど、勾配降下法の各反復の計算コストは高くなるため、そのような場合には確率的勾配降下法が好まれる。
- 非凸の場合、チェックすべき局所最小値の数が指数的になりうるため、確率的勾配降下法に対する最適性保証は一般には得られません。

15.4.6 演習

1. 確率的勾配降下法について、さまざまな学習率スケジュールとさまざまな反復回数を試してみよ。特に、最適解 $(0, 0)$ からの距離を反復回数の関数としてプロットせよ。
2. 関数 $f(x_1, x_2) = x_1^2 + 2x_2^2$ に対して、勾配に正規ノイズを加えることが、$\mathbf{x}$ が正規分布から引かれるときの損失関数 $f(\mathbf{x}, \mathbf{w}) = (x_1 - w_1)^2 + 2(x_2 - w_2)^2$ を最小化することと等価であることを証明せよ。
3. $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ から復元抽出する場合と非復元抽出する場合で、確率的勾配降下法の収束を比較せよ。
4. ある勾配（あるいはそれに対応するある座標）が他のすべての勾配より一貫して大きい場合、確率的勾配降下法ソルバをどのように変更するか。

5. $f(x) = x^2(1 + \sin x)$ と仮定する。f には局所最小値がいくつあるか。それを最小化するために、すべての局所最小値を評価する必要があるように f を変更できるか。

15.5 ミニバッチ確率的勾配降下法

これまで、勾配ベースの学習における 2 つの極端な方法を見てきた。Section 15.3 では、全データセットを使って勾配を計算し、1 回の走査ごとにパラメータを更新する。これに対して Section 15.4 では、1 つの訓練例を一度に処理して前進する。どちらにもそれぞれ欠点がある。データが非常に似通っている場合、勾配降下法は特に データ効率が良いわけではない。一方、確率的勾配降下法は、CPU や GPU がベクトル化の力を十分に活用できないため、特に 計算効率が良いわけではない。このことは、その中間に何かがあるはずだと示唆している。実際、これまで議論してきた例では、まさにその中間を使ってきた。

15.5.1 ベクトル化とキャッシュ

ミニバッチを使うかどうかの判断の核心にあるのは計算効率である。これは、複数の GPU や複数のサーバーへの並列化を考えると最も理解しやすい。この場合、少なくとも各 GPU に 1 枚の画像を送る必要がある。1 サーバーあたり 8 GPU、16 サーバーなら、ミニバッチサイズは少なくとも 128 になる。

単一 GPU、あるいは CPU の場合は少し事情が複雑である。これらのデバイスには複数種類のメモリがあり、しばしば複数種類の計算ユニットがあり、それらの間には異なる帯域幅制約がある。たとえば CPU には少数のレジスタがあり、その後に L1、L2、場合によっては L3 キャッシュ（これは異なるプロセッサコア間で共有される）がある。これらのキャッシュは、サイズとレイテンシが大きくなるにつれて、同時に帯域幅は小さくなる。要するに、プロセッサは主記憶インターフェースが供給できる量よりもはるかに多くの演算を実行できるのである。

まず、2GHz で 16 コア、AVX-512 ベクトル化を備えた CPU は、最大で $2 \cdot 10^9 \cdot 16 \cdot 32 = 10^{12}$ バイト/秒を処理できる。GPU の能力はこの数値を容易に 100 倍上回る。一方、中級クラスのサーバープロセッサでは 100 GB/s 程度の帯域幅しかないことがあり、つまりプロセッサに十分なデータを供給するのに必要な量の 10 分の 1 未満である。さらに悪いことに、すべてのメモリアクセスが同じではない。メモリインターフェースは通常 64 ビット幅以上（たとえば GPU では最大 384 ビット）なので、1 バイトを読むだけでもはるかに広いアクセスのコストがかかる。

第二に、最初のアクセスにはかなりのオーバーヘッドがあり、連続アクセスは
85 
比較的安価である（これはしばしばバースト読み出しと呼ばれる）。複数ソケット、チップレット、その他の構造がある場合のキャッシュなど、考慮すべき

ことはまだたくさんある。より詳しい議論については、この Wikipedia article[85] を参照されたい。

こうした制約を緩和する方法は、実際にプロセッサへデータを供給できるほど高速な CPU キャッシュの階層を使うことである。これこそが、深層学習におけるバッチ処理の 原動力である。話を簡単にするために、行列積、たとえば $\mathbf{A} = \mathbf{BC}$ を考える。$\mathbf{A}$ を計算する方法はいくつかある。たとえば、次のような選択肢がある。

1. $\mathbf{A}_{ij} = \mathbf{B}_{i,:}\mathbf{C}_{:,j}$ を計算する、つまり内積を用いて要素ごとに計算する。
2. $\mathbf{A}_{:,j} = \mathbf{BC}_{:,j}$ を計算する、つまり列ごとに計算する。同様に、$\mathbf{A}$ を 1 行 $\mathbf{A}_{i,:}$ ずつ計算することもできる。
3. 単純に $\mathbf{A} = \mathbf{BC}$ を計算する。
4. $\mathbf{B}$ と $\mathbf{C}$ をより小さなブロック行列に分割し、$\mathbf{A}$ を 1 ブロックずつ計算する。

最初の方法を採ると、要素 $\mathbf{A}_{ij}$ を計算するたびに、1 つの行ベクトルと 1 つの列ベクトルを CPU にコピーする必要がある。さらに悪いことに、行列要素は順番に並んでいるため、2 つのベクトルのうち一方については、メモリから読み出す際に多くの離れた場所へアクセスしなければならない。2 番目の方法のほうがはるかに有利である。この方法では、列ベクトル $\mathbf{C}_{:,j}$ を CPU キャッシュに保持したまま、$\mathbf{B}$ を走査し続けることができる。これによりメモリ帯域幅の要求が半分になり、その分アクセスも速くなる。もちろん、3 番目の方法が最も望ましい。しかし残念ながら、多くの行列はキャッシュに完全には収まらない（まさにそれがここで議論していることである）。とはいえ、4 番目の方法は実用上有用な代替案を提供する。つまり、行列のブロックをキャッシュに移し、局所的に掛け合わせることができる。最適化されたライブラリがこれを代わりに行ってくれる。これらの操作が実際にどれほど効率的か見てみよう。

計算効率に加えて、Python および深層学習フレームワーク自体によるオーバーヘッドもかなり大きい。コマンドを実行するたびに、Python インタプリタが MXNet エンジンへ命令を送り、それを計算グラフに挿入してスケジューリング時に処理する必要があることを思い出してほしい。このようなオーバーヘッドはかなり有害になりえる。要するに、可能な限りベクトル化（および行列）を使うことが強く推奨される。

```
%matplotlib inline
from d2l import torch as d2l
import numpy as np
import time
import torch
from torch import nn

A = torch.zeros(256, 256)
B = torch.randn(256, 256)
C = torch.randn(256, 256)
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
import time
npx.set_np()

A = np.zeros((256, 256))
B = np.random.normal(0, 1, (256, 256))
C = np.random.normal(0, 1, (256, 256))
```

```
[06:58:32] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
class Timer:  #@save
    """ 複数回の実行時間を記録する。"""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        """ タイマーを開始する。"""
        self.tik = time.time()

    def stop(self):
        """ タイマーを停止し、時間をリストに記録する。"""
        self.times.append(time.time() - self.tik)
        return self.times[-1]

    def avg(self):
        """ 平均時間を返す。"""
        return sum(self.times) / len(self.times)

    def sum(self):
        """ 時間の合計を返す。"""
        return sum(self.times)

    def cumsum(self):
        """ 累積時間を返す。"""
        return np.array(self.times).cumsum().tolist()

timer = Timer()
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import numpy as np
import tensorflow as tf
import time

A = tf.Variable(d2l.zeros((256, 256)))
B = tf.Variable(d2l.normal([256, 256], 0, 1))
C = tf.Variable(d2l.normal([256, 256], 0, 1))
```

この本の残りでは実行時間を頻繁にベンチマークするので、タイマーを定義しておこう。

```
class Timer:  #@save
    """ 複数回の実行時間を記録する。"""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        """ タイマーを開始する。"""
        self.tik = time.time()

    def stop(self):
        """ タイマーを停止し、時間をリストに記録する。"""
        self.times.append(time.time() - self.tik)
        return self.times[-1]

    def avg(self):
        """ 平均時間を返す。"""
        return sum(self.times) / len(self.times)

    def sum(self):
        """ 時間の合計を返す。"""
        return sum(self.times)

    def cumsum(self):
        """ 累積時間を返す。"""
        return np.array(self.times).cumsum().tolist()

timer = Timer()
```

```
class Timer:  #@save
    """ 複数回の実行時間を記録する。"""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        """ タイマーを開始する。"""
        self.tik = time.time()

    def stop(self):
        """ タイマーを停止し、時間をリストに記録する。"""
        self.times.append(time.time() - self.tik)
        return self.times[-1]

    def avg(self):
        """ 平均時間を返す。"""
        return sum(self.times) / len(self.times)

    def sum(self):
        """ 時間の合計を返す。"""
        return sum(self.times)
```

(continues on next page)

(continued from previous page)

```
    def cumsum(self):
        """ 累積時間を返す。"""
        return np.array(self.times).cumsum().tolist()

timer = Timer()
```

```
def train_sgd(lr, batch_size, num_epochs=2):
    data_iter, feature_dim = get_data_ch11(batch_size)
    return train_ch11(
        sgd, None, {'lr': lr}, data_iter, feature_dim, num_epochs)

gd_res = train_sgd(1, 1500, 10)
```

```
class Timer:  #@save
    """ 複数回の実行時間を記録する。"""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        """ タイマーを開始する。"""
        self.tik = time.time()

    def stop(self):
        """ タイマーを停止し、時間をリストに記録する。"""
        self.times.append(time.time() - self.tik)
        return self.times[-1]

    def avg(self):
        """ 平均時間を返す。"""
        return sum(self.times) / len(self.times)

    def sum(self):
        """ 時間の合計を返す。"""
        return sum(self.times)

    def cumsum(self):
        """ 累積時間を返す。"""
        return np.array(self.times).cumsum().tolist()

timer = Timer()
```

要素ごとの代入では、$\mathbf{B}$ と $\mathbf{C}$ のすべての行と列をそれぞれ走査して、値を $\mathbf{A}$ に代入する。

```
# Compute A = BC one element at a time
timer.start()
for i in range(256):
    for j in range(256):
```

(continues on next page)

(continued from previous page)

```
        A[i, j] = torch.dot(B[i, :], C[:, j])
timer.stop()
```

```
1.704789400100708
```

```
# Compute A = BC one element at a time
timer.start()
for i in range(256):
    for j in range(256):
        A[i, j] = np.dot(B[i, :], C[:, j])
A.wait_to_read()
timer.stop()
```

```
78.08700704574585
```

```
sgd_res = train_sgd(0.005, 1)
```

```
# Compute A = BC one element at a time
timer.start()
for i in range(256):
    for j in range(256):
        A[i, j].assign(tf.tensordot(B[i, :], C[:, j], axes=1))
timer.stop()
```

```
159.19349670410156
```

より高速な戦略は、列ごとに代入することである。

```
# Compute A = BC one column at a time
timer.start()
for j in range(256):
    A[:, j] = torch.mv(B, C[:, j])
timer.stop()
```

```
0.010866403579711914
```

```
# Compute A = BC one column at a time
timer.start()
for j in range(256):
    A[:, j] = np.dot(B, C[:, j])
A.wait_to_read()
timer.stop()
```

```
0.2190697193145752
```

```
mini1_res = train_sgd(.4, 100)
```

```
timer.start()
for j in range(256):
    A[:, j].assign(tf.tensordot(B, C[:, j], axes=1))
timer.stop()
```

```
0.48426294326782227
```

最後に、最も効果的なのは、全体の操作を 1 つのブロックで実行する方法である。任意の 2 つの行列 $\mathbf{B} \in \mathbb{R}^{m \times n}$ と $\mathbf{C} \in \mathbb{R}^{n \times p}$ の積には、スカラー乗算と加算を別々の演算として数えると（実際には融合されるが）、およそ $2mnp$ 回の浮動小数点演算が必要であることに注意されたい。したがって、256×256 の 2 つの行列を掛けるには 0.03 十億回の浮動小数点演算が必要である。それぞれの操作速度を見てみよう。

```
# Compute A = BC in one go
timer.start()
A = torch.mm(B, C)
timer.stop()

gigaflops = [0.03 / i for i in timer.times]
print(f'performance in Gigaflops: element {gigaflops[0]:.3f}, '
      f'column {gigaflops[1]:.3f}, full {gigaflops[2]:.3f}')
```

```
performance in Gigaflops: element 0.018, column 2.761, full 52.538
```

```
# Compute A = BC in one go
timer.start()
A = np.dot(B, C)
A.wait_to_read()
timer.stop()

gigaflops = [0.03 / i for i in timer.times]
print(f'performance in Gigaflops: element {gigaflops[0]:.3f}, '
      f'column {gigaflops[1]:.3f}, full {gigaflops[2]:.3f}')
```

```
performance in Gigaflops: element 0.000, column 0.137, full 1.426
```

```
mini2_res = train_sgd(.05, 10)
```

```
timer.start()
A.assign(tf.tensordot(B, C, axes=1))
timer.stop()

gigaflops = [0.03 / i for i in timer.times]
print(f'performance in Gigaflops: element {gigaflops[0]:.3f}, '
      f'column {gigaflops[1]:.3f}, full {gigaflops[2]:.3f}')
```

```
performance in Gigaflops: element 0.000, column 0.062, full 1.205
```

15.5.2 ミニバッチ

これまで、パラメータ更新には単一の観測ではなく ミニバッチのデータを読み込むものとして当然視してきた。ここで、その理由を簡単な説明する。単一の観測を処理するには、多数の単一の行列-ベクトル（あるいはベクトル-ベクトル）積を実行する必要があり、これはかなり高コストで、基盤となる深層学習フレームワークにかなりのオーバーヘッドを生じさせる。これは、データにネットワークを適用して評価する場合（しばしば推論と呼ばれる）にも、パラメータ更新のために勾配を計算する場合にも当てはまる。つまり、$\mathbf{w} \leftarrow \mathbf{w} - \eta_t \mathbf{g}_t$ を実行するたびに当てはまり、ここで

$$\mathbf{g}_t = \partial_{\mathbf{w}} f(\mathbf{x}_t, \mathbf{w}) \tag{15.5.1}$$

である。

この操作の 計算効率は、複数の観測をまとめたミニバッチに対して一度に適用することで高められる。つまり、単一の観測に対する勾配 $\mathbf{g}_t$ を、小さなバッチに対する次の勾配で置き換える。

$$\mathbf{g}_t = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}) \tag{15.5.2}$$

これが $\mathbf{g}_t$ の統計的性質にどう影響するか見てみよう。$\mathbf{x}_t$ もミニバッチ $\mathcal{B}_t$ のすべての要素も訓練セットから一様ランダムに抽出されるので、勾配の期待値は変わらない。一方、分散は大幅に減少する。ミニバッチ勾配は、平均化される $b \stackrel{\text{def}}{=} |\mathcal{B}_t|$ 個の独立な勾配から構成されるため、その標準偏差は $b^{-\frac{1}{2}}$ 倍に減少する。これ自体は良いことである。なぜなら、更新が全勾配とより確実に整合することを意味するからである。

素朴に考えると、大きなミニバッチ $\mathcal{B}_t$ を選ぶことが常に望ましいように見える。ところが、ある点を超えると、標準偏差の追加的な減少は、計算コストの線形増加に比べてごくわずかになる。実際には、良い計算効率を得られ、しかも GPU のメモリに収まる程度に十分大きいミニバッチを選ぶ。節約効果を示すために、コードを見てみよう。ここでは同じ行列積を実行するが、今度は 64 列ずつの「ミニバッチ」に分割して計算する。

```
timer.start()
for j in range(0, 256, 64):
    A[:, j:j+64] = torch.mm(B, C[:, j:j+64])
timer.stop()
print(f'performance in Gigaflops: block {0.03 / timer.times[3]:.3f}')
```

```
performance in Gigaflops: block 29.579
```

```
timer.start()
for j in range(0, 256, 64):
    A[:, j:j+64] = np.dot(B, C[:, j:j+64])
timer.stop()
print(f'performance in Gigaflops: block {0.03 / timer.times[3]:.3f}')
```

```
performance in Gigaflops: block 2.848
```

```
d2l.set_figsize([6, 3])
d2l.plot(*list(map(list, zip(gd_res, sgd_res, mini1_res, mini2_res))),
         'time (sec)', 'loss', xlim=[1e-2, 10],
         legend=['gd', 'sgd', 'batch size=100', 'batch size=10'])
d2l.plt.gca().set_xscale('log')
```

```
timer.start()
for j in range(0, 256, 64):
    A[:, j:j+64].assign(tf.tensordot(B, C[:, j:j+64], axes=1))
timer.stop()
print(f'performance in Gigaflops: block {0.03 / timer.times[3]:.3f}')
```

```
performance in Gigaflops: block 2.942
```

見てわかるように、ミニバッチ上での計算は、実質的に全行列上での計算と同じくらい効率的である。ここで注意が必要である。Section 11.5 では、ミニバッチ内の分散の大きさに強く依存する正則化を使った。ミニバッチサイズを大きくすると分散は小さくなり、それに伴ってバッチ正規化によるノイズ注入の利点も小さくなる。適切な項の再スケーリングと計算方法については、たとえば Ioffe (2017) を参照されたい。

15.5.3 データセットの読み込み



86 ミニバッチがデータからどのように効率よく生成されるか見てみよう。以下では、異なる航空機の翼の 騒音[86] をテストするために NASA が開発したデータセットを使って、これらの最適化アルゴリズムを比較する。便宜上、最初の 1,500 例だけを使う。前処理としてデータはホワイトニングされており、つまり平均を引き、各座標の分散を 1 に再スケーリングしている。

```
#@save
d2l.DATA_HUB['airfoil'] = (d2l.DATA_URL + 'airfoil_self_noise.dat',
                           '76e5be1548fd8222e5074cf0faae75edff8cf93f')

#@save
def get_data_ch11(batch_size=10, n=1500):
    data = np.genfromtxt(d2l.download('airfoil'),
                         dtype=np.float32, delimiter='\t')
    data = torch.from_numpy((data - data.mean(axis=0)) / data.std(axis=0))
    data_iter = d2l.load_array((data[:n, :-1], data[:n, -1]),
                               batch_size, is_train=True)
    return data_iter, data.shape[1]-1
```

```
#@save
d2l.DATA_HUB['airfoil'] = (d2l.DATA_URL + 'airfoil_self_noise.dat',
                           '76e5be1548fd8222e5074cf0faae75edff8cf93f')

#@save
def get_data_ch11(batch_size=10, n=1500):
    data = np.genfromtxt(d2l.download('airfoil'),
                         dtype=np.float32, delimiter='\t')
    data = (data - data.mean(axis=0)) / data.std(axis=0)
    data_iter = d2l.load_array(
        (data[:n, :-1], data[:n, -1]), batch_size, is_train=True)
    return data_iter, data.shape[1]-1
```

```
#@save
d2l.DATA_HUB['airfoil'] = (d2l.DATA_URL + 'airfoil_self_noise.dat',
                           '76e5be1548fd8222e5074cf0faae75edff8cf93f')

#@save
def get_data_ch11(batch_size=10, n=1500):
    data = np.genfromtxt(d2l.download('airfoil'),
                         dtype=np.float32, delimiter='\t')
    data = (data - data.mean(axis=0)) / data.std(axis=0)
    data_iter = d2l.load_array((data[:n, :-1], data[:n, -1]),
                               batch_size, is_train=True)
    return data_iter, data.shape[1]-1
```

15.5.4 ゼロからの実装

Section 6.4 におけるミニバッチ確率的勾配降下法の実装を思い出してほしい。以下では、少し一般化した実装を示す。便宜上、後のこの章で導入する他の最適化アルゴリズムと同じ呼び出しシグネチャにしている。具体的には、状態入力 `states` を追加し、ハイパーパラメータを辞書 `hyperparams` に入れる。さらに、訓練関数では各ミニバッチ例の損失を平均するので、最適化アルゴリズム内で勾配をバッチサイズで割る必要はない。

```
def sgd(params, states, hyperparams):
    for p in params:
```

(continues on next page)

(continued from previous page)

```
        p.data.sub_(hyperparams['lr'] * p.grad)
        p.grad.data.zero_()
```

```
def sgd(params, states, hyperparams):
    for p in params:
        p[:] -= hyperparams['lr'] * p.grad
```

```
def sgd(params, grads, states, hyperparams):
    for param, grad in zip(params, grads):
        param.assign_sub(hyperparams['lr']*grad)
```

次に、この章で後ほど導入する他の最適化アルゴリズムの利用を容易にするため、汎用的な訓練関数を実装する。これは線形回帰モデルを初期化し、ミニバッチ確率的勾配降下法や、その後に導入する他のアルゴリズムでモデルを訓練するために使える。

```
#@save
def train_ch11(trainer_fn, states, hyperparams, data_iter,
               feature_dim, num_epochs=2):
    # 初期化
    w = torch.normal(mean=0.0, std=0.01, size=(feature_dim, 1),
                     requires_grad=True)
    b = torch.zeros((1), requires_grad=True)
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    # 訓練
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            l = loss(net(X), y).mean()
            l.backward()
            trainer_fn([w, b], states, hyperparams)
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.sum()/num_epochs:.3f} sec/
↪epoch')
    return timer.cumsum(), animator.Y[0]
```

```
#@save
def train_ch11(trainer_fn, states, hyperparams, data_iter,
               feature_dim, num_epochs=2):
    # 初期化
    w = np.random.normal(scale=0.01, size=(feature_dim, 1))
    b = np.zeros(1)
```

(continues on next page)

(continued from previous page)

```
    w.attach_grad()
    b.attach_grad()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    # 訓練
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            with autograd.record():
                l = loss(net(X), y).mean()
            l.backward()
            trainer_fn([w, b], states, hyperparams)
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.sum()/num_epochs:.3f} sec/
↪epoch')
    return timer.cumsum(), animator.Y[0]
```

```
#@save
def train_ch11(trainer_fn, states, hyperparams, data_iter,
               feature_dim, num_epochs=2):
    # 初期化
    w = tf.Variable(tf.random.normal(shape=(feature_dim, 1),
                                     mean=0, stddev=0.01),trainable=True)
    b = tf.Variable(tf.zeros(1), trainable=True)

    # 訓練
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()

    for _ in range(num_epochs):
        for X, y in data_iter:
          with tf.GradientTape() as g:
            l = tf.math.reduce_mean(loss(net(X), y))

          dw, db = g.gradient(l, [w, b])
          trainer_fn([w, b], [dw, db], states, hyperparams)
          n += X.shape[0]
          if n % 200 == 0:
              timer.stop()
              p = n/X.shape[0]
              q = p/tf.data.experimental.cardinality(data_iter).numpy()
              r = (d2l.evaluate_loss(net, data_iter, loss),)
              animator.add(q, r)
              timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.sum()/num_epochs:.3f} sec/
```

(continues on next page)

(continued from previous page)

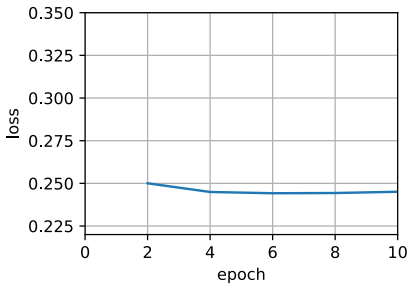
```
↪epoch')
    return timer.cumsum(), animator.Y[0]
```

バッチ勾配降下法で最適化がどのように進むか見てみよう。これはミニバッチサイズを 1500（つまり例の総数）に設定することで実現できる。その結果、モデルパラメータは各エポックで 1 回しか更新されない。進展はほとんどない。実際、6 ステップ後には進展が止まる。

```
def train_sgd(lr, batch_size, num_epochs=2):
    data_iter, feature_dim = get_data_ch11(batch_size)
    return train_ch11(
        sgd, None, {'lr': lr}, data_iter, feature_dim, num_epochs)

gd_res = train_sgd(1, 1500, 10)
```

```
loss: 0.245, 0.027 sec/epoch
```

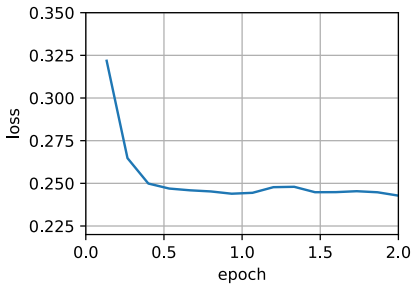


バッチサイズが 1 に等しいとき、最適化には確率的勾配降下法を使う。実装を簡単にするため、一定の（とはいえ小さな）学習率を選んだ。確率的勾配降下法では、例が処理されるたびにモデルパラメータが更新される。この場合、1 エポックあたり 1500 回の更新に相当する。見てわかるように、目的関数の値の低下は 1 エポック後に鈍化する。1 エポック内で両方の手続きが 1500 例を処理したにもかかわらず、この実験では確率的勾配降下法のほうが勾配降下法よりも時間を要する。これは、確率的勾配降下法がより頻繁にパラメータを更新し、単一の観測を 1 つずつ処理するのは効率が悪いためである。

```
sgd_res = train_sgd(0.005, 1)
```

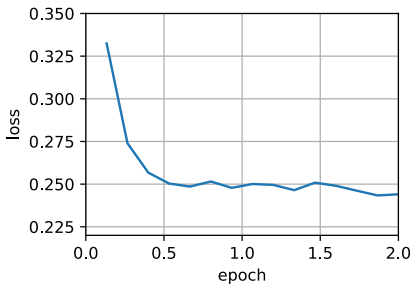
```
loss: 0.243, 0.750 sec/epoch
```

最後に、バッチサイズが 100 に等しいとき、最適化にはミニバッチ確率的勾配降下法を使う。1 エポックあたりに必要な時間は、確率的勾配降下法やバッチ勾配降下法より短くなる。




```
mini1_res = train_sgd(.4, 100)
```

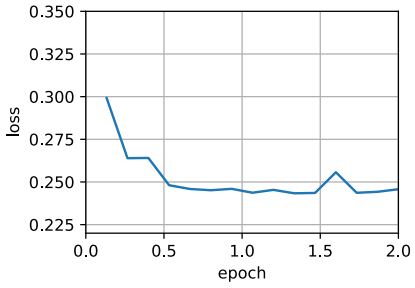
```
loss: 0.244, 0.024 sec/epoch
```



バッチサイズを 10 に減らすと、各バッチの処理効率が下がるため、各エポックの時間は増加する。

```
mini2_res = train_sgd(.05, 10)
```

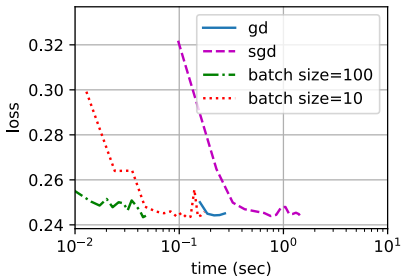
```
loss: 0.246, 0.087 sec/epoch
```



これで、先の 4 つの実験について時間と損失を比較できる。見てわかるように、確率的勾配降下法は処理した例の数という観点では GD より速く収束する

が、勾配を例ごとに計算するのは効率的ではないため、同じ損失に到達するまでにより多くの時間を要する。ミニバッチ確率的勾配降下法は、収束速度と計算効率のトレードオフをうまく取ることができる。バッチサイズ 10 は確率的勾配降下法より効率的であり、バッチサイズ 100 は実行時間の観点では GD を上回る。

```
d2l.set_figsize([6, 3])
d2l.plot(*list(map(list, zip(gd_res, sgd_res, mini1_res, mini2_res))),
         'time (sec)', 'loss', xlim=[1e-2, 10],
         legend=['gd', 'sgd', 'batch size=100', 'batch size=10'])
d2l.plt.gca().set_xscale('log')
```



15.5.5 簡潔な実装

Gluon では、Trainer クラスを使って最適化アルゴリズムを呼び出せる。これは汎用的な訓練関数を実装するために使われる。これはこの章を通して使う。

```
#@save
def train_concise_ch11(trainer_fn, hyperparams, data_iter, num_epochs=4):
    # 初期化
    net = nn.Sequential(nn.Linear(5, 1))
    def init_weights(module):
        if type(module) == nn.Linear:
            torch.nn.init.normal_(module.weight, std=0.01)
    net.apply(init_weights)

    optimizer = trainer_fn(net.parameters(), **hyperparams)
    loss = nn.MSELoss(reduction='none')
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            optimizer.zero_grad()
            out = net(X)
            y = y.reshape(out.shape)
            l = loss(out, y)
            l.mean().backward()
            optimizer.step()
```

(continues on next page)

(continued from previous page)

```
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                # `MSELoss` computes squared error without the 1/2 factor
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss) / 2,))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.sum()/num_epochs:.3f} sec/
↪epoch')
```

```
#@save
def train_concise_ch11(tr_name, hyperparams, data_iter, num_epochs=2):
    # 初期化
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init.Normal(sigma=0.01))
    trainer = gluon.Trainer(net.collect_params(), tr_name, hyperparams)
    loss = gluon.loss.L2Loss()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(X.shape[0])
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.sum()/num_epochs:.3f} sec/
↪epoch')
```

```
#@save
def train_concise_ch11(trainer_fn, hyperparams, data_iter, num_epochs=2):
    # 初期化
    net = tf.keras.Sequential()
    net.add(tf.keras.layers.Dense(1,
            kernel_initializer=tf.random_normal_initializer(stddev=0.01)))
    optimizer = trainer_fn(**hyperparams)
    loss = tf.keras.losses.MeanSquaredError()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            with tf.GradientTape() as g:
                out = net(X)
                l = loss(y, out)
                params = net.trainable_variables
```

(continues on next page)

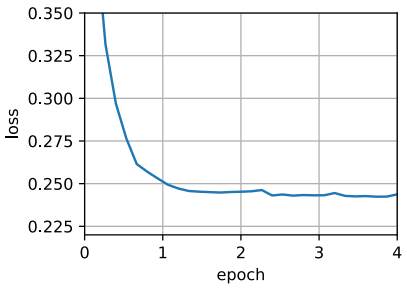
(continued from previous page)

```
                grads = g.gradient(l, params)
            optimizer.apply_gradients(zip(grads, params))
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                p = n/X.shape[0]
                q = p/tf.data.experimental.cardinality(data_iter).numpy()
                # `MeanSquaredError` computes squared error without the 1/2
                # factor
                r = (d2l.evaluate_loss(net, data_iter, loss) / 2,)
                animator.add(q, r)
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.sum()/num_epochs:.3f} sec/
↪epoch')
```

Gluon を使って最後の実験を繰り返すと、同じ挙動が得られる。

```
data_iter, _ = get_data_ch11(10)
trainer = torch.optim.SGD
train_concise_ch11(trainer, {'lr': 0.01}, data_iter)
```

```
loss: 0.244, 0.101 sec/epoch
```



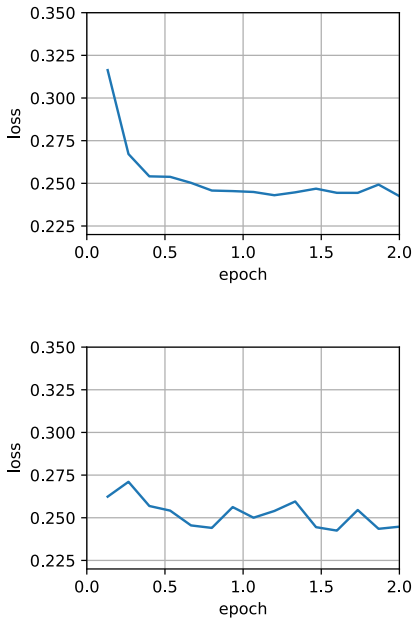

```
data_iter, _ = get_data_ch11(10)
train_concise_ch11('sgd', {'learning_rate': 0.05}, data_iter)
```

```
loss: 0.243, 0.341 sec/epoch
```

```
data_iter, _ = get_data_ch11(10)
trainer = tf.keras.optimizers.SGD
train_concise_ch11(trainer, {'learning_rate': 0.05}, data_iter)
```

```
loss: 0.245, 1.258 sec/epoch
```

15.5.6 まとめ



- ベクトル化により、深層学習フレームワーク由来のオーバーヘッドが減り、CPU や GPU でのメモリ局所性とキャッシュ利用が改善されるため、コードはより効率的になる。
- 確率的勾配降下法に由来する統計効率と、データを大きなバッチでまとめて処理することに由来する計算効率の間にはトレードオフがある。
- ミニバッチ確率的勾配降下法は、計算効率と統計効率の両方の利点を兼ね備えている。
- ミニバッチ確率的勾配降下法では、訓練データのランダムな順列から得られたバッチを処理する（つまり、各観測は各エポックで 1 回だけ、ただしランダムな順序で処理される）。
- 訓練中に学習率を減衰させることが推奨される。
- 一般に、ミニバッチ確率的勾配降下法は、時計時間で測ったとき、より小さなリスクへ収束するまでの速さにおいて、確率的勾配降下法や勾配降下法より高速である。

15.5.7 演習

1. バッチサイズと学習率を変更し、目的関数の値の低下速度と各エポックで消費される時間を観察せよ。
2. MXNet のドキュメントを読み、`Trainer` クラスの `set_learning_rate` 関数を使って、各エポック後にミニバッチ確率的勾配降下法の学習率を前回の 1/10 にせよ。

3. ミニバッチ確率的勾配降下法と、実際に訓練セットから 復元抽出する変種を比較せよ。何が起こるか。

4. 悪意ある魔神が、あなたに知らせずにデータセットを複製したとする（つまり、各観測が 2 回ずつ現れ、データセットの大きさは元の 2 倍になるが、誰もそれを教えてくれない）。確率的勾配降下法、ミニバッチ確率的勾配降下法、勾配降下法の挙動はどう変わるか。

15.6 モメンタム

Section 15.4 では、確率的勾配降下法を行うとき、すなわち勾配のノイズを含む変種しか利用できない最適化を行うときに何が起こるかを復習した。特に、ノイズのある勾配では、ノイズに直面したときの学習率の選び方にいっそう注意が必要であることに気づいた。これを急激に下げすぎると、収束が止まってしまう。逆に寛大すぎると、ノイズが最適解から私たちを押し戻し続けるため、十分によい解に収束できない。

15.6.1 基礎

この節では、実際によく現れるある種の最適化問題に対して、より効果的な最適化アルゴリズムを探る。

漏れのある平均

前節では、計算を高速化する手段としてミニバッチ SGD について議論した。また、勾配を平均することで分散が減るというううれしい副作用もあった。ミニバッチ確率的勾配降下法は次のように計算できる。

$$\mathbf{g}_{t,t-1} = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}_{t-1}) = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \mathbf{h}_{i,t-1}. \tag{15.6.1}$$

記法を簡単にするため、ここでは $\mathbf{h}_{i,t-1} = \partial_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w}_{t-1})$ を、時刻 $t-1$ に更新された重みを用いたサンプル i の確率的勾配降下法として用いた。ミニバッチ上で勾配を平均すること以上に、分散低減の効果を活かせたら理想的である。この目的を達成する一つの方法は、勾配計算を「漏れのある平均」に置き換えることである。

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1} \tag{15.6.2}$$

ここで $\beta \in (0, 1)$ とする。これは実質的に、その時点の勾配を複数の過去の勾配で平均したものに置き換えることを意味する。$\mathbf{v}$ は速度と呼ばれる。これは、重い球が目的関数の地形を転がり落ちるときに過去の力を積分するのと同様に、過去の勾配を蓄積する。何が起きているかをより詳しく見るために、$\mathbf{v}_t$ を再帰

的に展開すると

$$\mathbf{v}_t = \beta^2\mathbf{v}_{t-2} + \beta\mathbf{g}_{t-1,t-2} + \mathbf{g}_{t,t-1} = \ldots, = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau,t-\tau-1}. \tag{15.6.3}$$

β が大きいと長い範囲の平均になり、小さいと勾配法に対するわずかな補正にとどまる。新しい勾配の置き換えは、もはや特定の 1 回の勾配降下で最急降下方向を指すのではなく、過去の勾配の重み付き平均の方向を指す。これにより、実際にその上で勾配を計算するコストを払わずに、バッチで平均することの利点の大部分を実現できる。この平均化手続きについては、後でさらに詳しく見直す。

上の考え方は、現在では加速勾配法として知られる手法、たとえばモメンタム付き勾配法の基礎となった。これらは、最適化問題が悪条件である場合、すなわちある方向では他の方向より進みがずっと遅く、細い峡谷のように見える場合に、はるかに効果的であるという追加の利点を持つ。さらに、連続する勾配を平均することで、より安定した降下方向を得られる。実際、ノイズのない凸問題であっても加速が効くという点は、モメンタムが機能する重要な理由の一つであり、また非常によく効く理由でもある。

予想されるように、その有効性ゆえにモメンタムは深層学習およびそれ以外の最適化でも広く研究されている。詳細な解析と対話的アニメーションについては、たとえば Goh (2017) による美しい解説記事[87] を参照されたい。これは Polyak (1964) によって提案された。Nesterov (2018) には、凸最適化の文脈での詳細な理論的議論がある。深層学習におけるモメンタムが有益であることは、以前から知られている。詳細は、たとえば Sutskever *et al.* (2013) の議論を参照されたい。



[87] Polyak (1964) によって提案された。Nesterov (2018) には、凸最適化の文脈での詳細な理論的議論がある。深層学習におけるモメンタムが有益であることは、以前から知られている。詳細は、たとえば Sutskever *et al.* (2013) の議論を参照されたい。

悪条件の問題

モメンタム法の幾何学的性質をよりよく理解するために、勾配降下法を再び取り上げる。ただし、今回はかなり扱いにくい目的関数を用いる。Section 15.3 では $f(\mathbf{x}) = x_1^2 + 2x_2^2$、すなわち中程度に歪んだ楕円体の目的関数を用いたことを思い出そう。ここでは、この関数を x_1 方向に引き伸ばすことでさらに歪める。

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2. \tag{15.6.4}$$

前と同様に、f の最小値は $(0, 0)$ にある。この関数は x_1 方向では非常に平坦である。では、この新しい関数に対して前と同じように勾配降下法を行うと何が起こるか見てみよう。学習率は 0.4 とする。

```
%matplotlib inline
from d2l import torch as d2l
import torch

eta = 0.4
def f_2d(x1, x2):
```

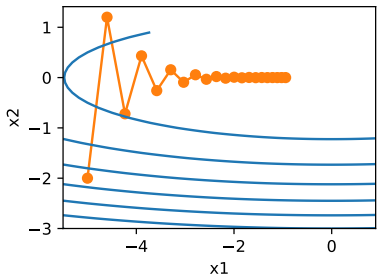
(continues on next page)

(continued from previous page)

```
    return 0.1 * x1 ** 2 + 2 * x2 ** 2
def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
epoch 20, x1: -0.943467, x2: -0.000073
```

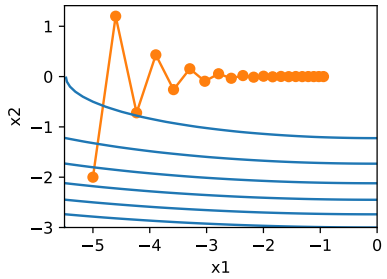



```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()

eta = 0.4
def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2
def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
epoch 20, x1: -0.943467, x2: -0.000073
[07:15:33] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```



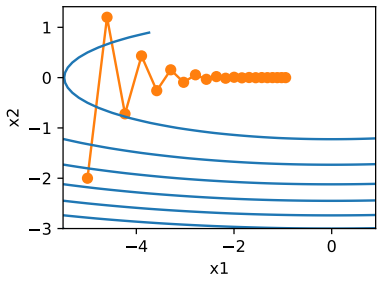
```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf

eta = 0.4
def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2
def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
epoch 20, x1: -0.943467, x2: -0.000073
```



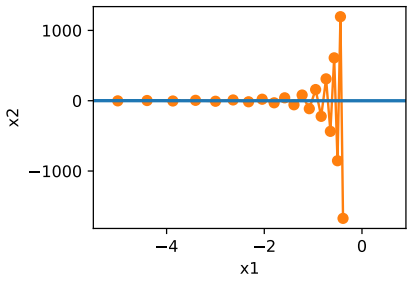
構成上、x_2 方向の勾配ははるかに大きく、水平な x_1 方向よりもずっと急速に変化する。したがって、私たちは望ましくない二つの選択肢の間に挟まれる。小さい学習率を選べば x_2 方向で解が発散しないことは保証できるが、x_1 方向での収束は遅くなる。逆に大きい学習率では x_1 方向では速く進むが、x_2 方向で発散する。以下の例は、学習率を 0.4 から 0.6 に少し上げただけでも何が起こるかを示している。x_1 方向の収束は改善するが、全体としての解の質はかなり悪くなる。

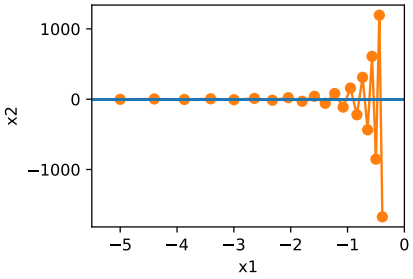
```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
epoch 20, x1: -0.387814, x2: -1673.365109
```

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
epoch 20, x1: -0.387814, x2: -1673.365109
```



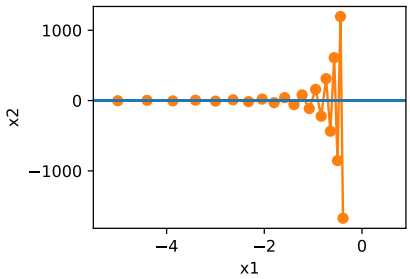



```
def momentum_2d(x1, x2, v1, v2):
    v1 = beta * v1 + 0.2 * x1
    v2 = beta * v2 + 4 * x2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

eta, beta = 0.6, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
epoch 20, x1: -0.387814, x2: -1673.365109
```



モメンタム法

モメンタム法を用いれば、上で述べた勾配降下法の問題を解決できる。上の最適化の軌跡を見ると、過去の勾配を平均するとうまくいきそうだと直感できるかもしれない。実際、x_1 方向では、よく整列した勾配が蓄積されるため、各ステップで進める距離が増える。逆に、勾配が振動する x_2 方向では、振動が互いに打ち消し合うため、集約された勾配によってステップサイズが小さくなる。勾配 $\mathbf{g}_t$ の代わりに $\mathbf{v}_t$ を用いると、次の更新式が得られる。

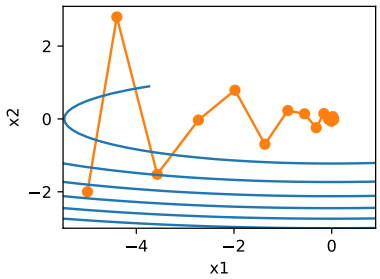
$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1},\\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t.\end{aligned} \tag{15.6.5}$$

$\beta = 0$ のとき、通常の勾配降下法に戻ることに注意しよう。数学的性質をさらに深く掘り下げる前に、まずは実際の挙動を簡単に見てみよう。

```
def momentum_2d(x1, x2, v1, v2):
    v1 = beta * v1 + 0.2 * x1
    v2 = beta * v2 + 4 * x2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

eta, beta = 0.6, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
epoch 20, x1: 0.007188, x2: 0.002553
```

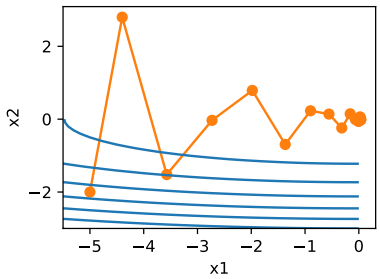



```
def momentum_2d(x1, x2, v1, v2):
    v1 = beta * v1 + 0.2 * x1
    v2 = beta * v2 + 4 * x2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

eta, beta = 0.6, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
epoch 20, x1: 0.007188, x2: 0.002553
```

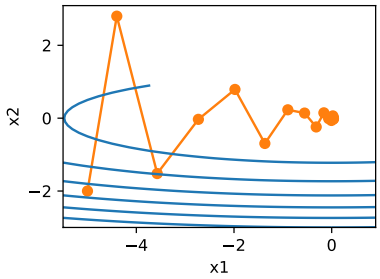

```
eta, beta = 0.6, 0.25
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```




```
def momentum_2d(x1, x2, v1, v2):
    v1 = beta * v1 + 0.2 * x1
    v2 = beta * v2 + 4 * x2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

eta, beta = 0.6, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
epoch 20, x1: 0.007188, x2: 0.002553
```

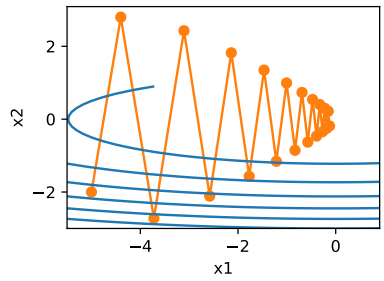


見てわかるように、以前と同じ学習率でも、モメンタムは依然としてうまく収束する。モメンタムのハイパーパラメータを下げるとどうなるか見てみよう。これを半分の $\beta = 0.25$ にすると、ほとんど収束しない軌跡になる。それでも、モメンタムなしの場合（そのときは解が発散する）よりはかなりよい。

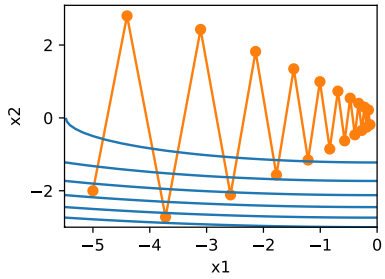
```
eta, beta = 0.6, 0.25
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
epoch 20, x1: -0.126340, x2: -0.186632
```

```
eta, beta = 0.6, 0.25
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```



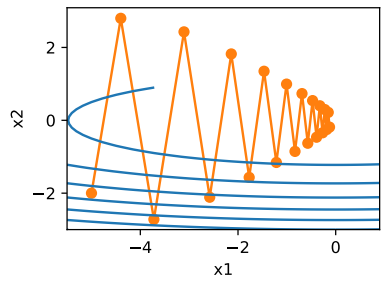

```
epoch 20, x1: -0.126340, x2: -0.186632
```




```
d2l.set_figsize()
betas = [0.95, 0.9, 0.6, 0]
for beta in betas:
    x = d2l.numpy(d2l.arange(40))
    d2l.plt.plot(x, beta ** x, label=f'beta = {beta:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```

```
eta, beta = 0.6, 0.25
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
epoch 20, x1: -0.126340, x2: -0.186632
```

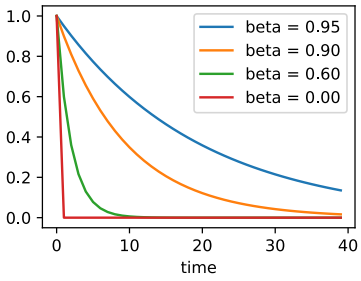


モメンタムは確率的勾配降下法、とくにミニバッチ確率的勾配降下法と組み合わせられることに注意しよう。その場合に変わるのは、勾配 $\mathbf{g}_{t,t-1}$ を $\mathbf{g}_t$ に置き換えることだけである。最後に、便宜上、時刻 $t = 0$ で $\mathbf{v}_0 = 0$ と初期化する。では、この漏れのある平均が実際に更新に対して何をしているのか見てみよう。

有効サンプル重み

$\mathbf{v}_t = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau,t-\tau-1}$ を思い出そう。極限では、項は $\sum_{\tau=0}^{\infty} \beta^\tau = \frac{1}{1-\beta}$ に足し合わさる。言い換えると、勾配降下法や確率的勾配降下法でサイズ η の一歩を踏み出す代わりに、サイズ $\frac{\eta}{1-\beta}$ の一歩を踏み出しつつ、同時に、より扱いやすい降下方向を扱っていることになる。これは一度に二つの利点である。β の異なる選び方によって重み付けがどう変わるかを示すために、下の図を考えよう。

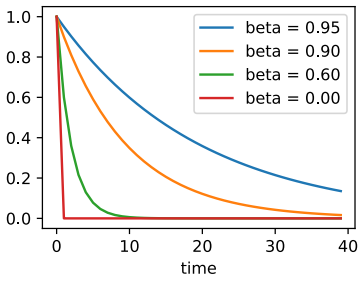
```
d2l.set_figsize()
betas = [0.95, 0.9, 0.6, 0]
for beta in betas:
    x = d2l.numpy(d2l.arange(40))
    d2l.plt.plot(x, beta ** x, label=f'beta = {beta:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```



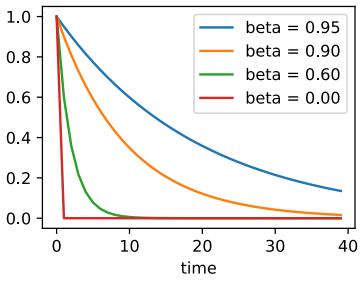

```
d2l.set_figsize()
betas = [0.95, 0.9, 0.6, 0]
for beta in betas:
    x = d2l.numpy(d2l.arange(40))
    d2l.plt.plot(x, beta ** x, label=f'beta = {beta:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```


```
def train_momentum(lr, momentum, num_epochs=2):
    d2l.train_ch11(sgd_momentum, init_momentum_states(feature_dim),
                   {'lr': lr, 'momentum': momentum}, data_iter,
                   feature_dim, num_epochs)

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
train_momentum(0.02, 0.5)
```




```
d2l.set_figsize()
betas = [0.95, 0.9, 0.6, 0]
for beta in betas:
    x = d2l.numpy(d2l.arange(40))
    d2l.plt.plot(x, beta ** x, label=f'beta = {beta:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```



15.6.2 実践的な実験

実際にモメンタムがどのように働くか、すなわち適切な最適化器の文脈で使ったときにどうなるかを見てみよう。そのためには、もう少しスケーラブルな実装が必要である。

ゼロからの実装

（ミニバッチ）確率的勾配降下法と比べると、モメンタム法では補助変数、すなわち速度を保持する必要がある。これは勾配（および最適化問題の変数）と同じ形状を持つ。以下の実装では、これらの変数を `states` と呼ぶ。

```
def init_momentum_states(feature_dim):
    v_w = d2l.zeros((feature_dim, 1))
    v_b = d2l.zeros(1)
    return (v_w, v_b)
```

```
def init_momentum_states(feature_dim):
    v_w = d2l.zeros((feature_dim, 1))
    v_b = d2l.zeros(1)
    return (v_w, v_b)
```

```
train_momentum(0.01, 0.9)
```

```
def init_momentum_states(features_dim):
    v_w = tf.Variable(d2l.zeros((features_dim, 1)))
    v_b = tf.Variable(d2l.zeros(1))
    return (v_w, v_b)
```

```
def sgd_momentum(params, states, hyperparams):
    for p, v in zip(params, states):
        with torch.no_grad():
            v[:] = hyperparams['momentum'] * v + p.grad
            p[:] -= hyperparams['lr'] * v
        p.grad.data.zero_()
```

```
def sgd_momentum(params, states, hyperparams):
    for p, v in zip(params, states):
        v[:] = hyperparams['momentum'] * v + p.grad
        p[:] -= hyperparams['lr'] * v
```

```
train_momentum(0.005, 0.9)
```

```
def sgd_momentum(params, grads, states, hyperparams):
    for p, v, g in zip(params, states, grads):
            v[:].assign(hyperparams['momentum'] * v + g)
            p[:].assign(p - hyperparams['lr'] * v)
```

実際にこれがどう働くか見てみよう。

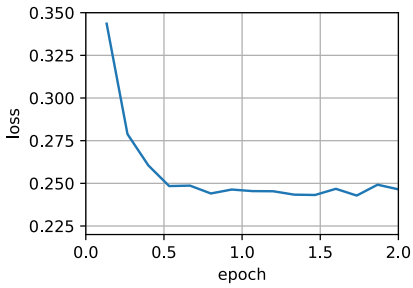
```
def train_momentum(lr, momentum, num_epochs=2):
    d2l.train_ch11(sgd_momentum, init_momentum_states(feature_dim),
                   {'lr': lr, 'momentum': momentum}, data_iter,
                   feature_dim, num_epochs)

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
train_momentum(0.02, 0.5)
```

```
loss: 0.246, 0.168 sec/epoch
```

```
def train_momentum(lr, momentum, num_epochs=2):
    d2l.train_ch11(sgd_momentum, init_momentum_states(feature_dim),
```

(continues on next page)

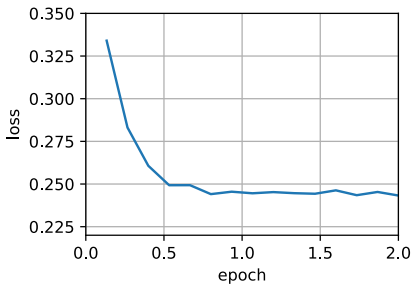


(continued from previous page)

```
                    {'lr': lr, 'momentum': momentum}, data_iter,
                    feature_dim, num_epochs)

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
train_momentum(0.02, 0.5)
```

```
loss: 0.243, 1.227 sec/epoch
```

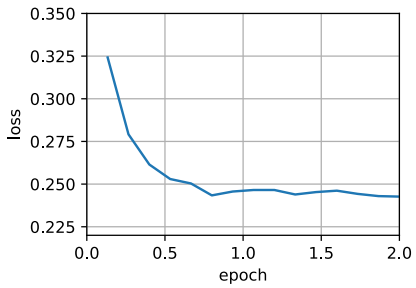



```
lambdas = [0.1, 1, 10, 19]
eta = 0.1
d2l.set_figsize((6, 4))
for lam in lambdas:
    t = d2l.numpy(d2l.arange(20))
    d2l.plt.plot(t, (1 - eta * lam) ** t, label=f'lambda = {lam:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```

```
def train_momentum(lr, momentum, num_epochs=2):
    d2l.train_ch11(sgd_momentum, init_momentum_states(feature_dim),
                    {'lr': lr, 'momentum': momentum}, data_iter,
                    feature_dim, num_epochs)

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
train_momentum(0.02, 0.5)
```

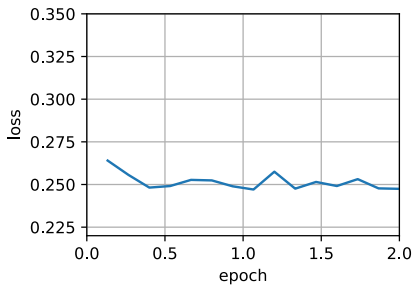
```
loss: 0.243, 1.148 sec/epoch
```



モメンタムのハイパーパラメータ momentum を 0.9 に増やすと、有効サンプルサイズは $\frac{1}{1-0.9} = 10$ とかなり大きくなる。制御しやすくするため、学習率は少し下げて 0.01 にする。

```
train_momentum(0.01, 0.9)
```

```
loss: 0.247, 0.140 sec/epoch
```



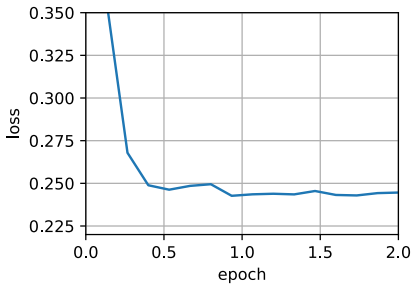
学習率をさらに下げると、滑らかでない最適化問題に関する問題にも対処できる。これを 0.005 にすると、よい収束特性が得られる。

```
train_momentum(0.005, 0.9)
```

```
loss: 0.245, 0.133 sec/epoch
```

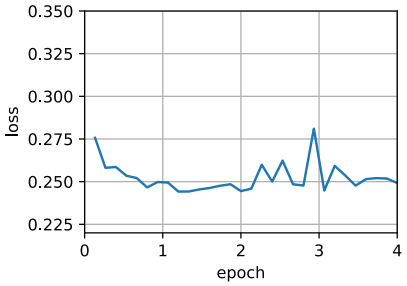
簡潔な実装

Gluon では、標準の sgd ソルバにすでにモメンタムが組み込まれているので、やることはほとんどない。対応するパラメータを設定すると、非常によく似た軌跡が得られる。



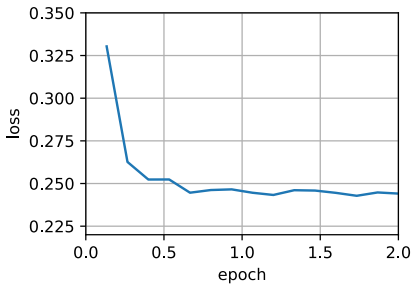

```
trainer = torch.optim.SGD
d2l.train_concise_ch11(trainer, {'lr': 0.005, 'momentum': 0.9}, data_iter)
```

```
loss: 0.249, 0.130 sec/epoch
```




```
d2l.train_concise_ch11('sgd', {'learning_rate': 0.005, 'momentum': 0.9},
                       data_iter)
```

```
loss: 0.244, 0.887 sec/epoch
```



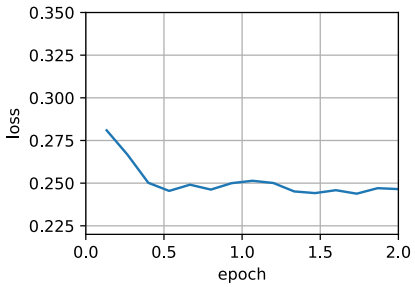

```
trainer = tf.keras.optimizers.SGD
```

(continues on next page)

(continued from previous page)

```
d2l.train_concise_ch11(trainer, {'learning_rate': 0.005, 'momentum': 0.9},
                       data_iter)
```

```
loss: 0.247, 1.224 sec/epoch
```



15.6.3 理論解析

ここまで、$f(x) = 0.1x_1^2 + 2x_2^2$ の 2 次元例はやや作為的に見えた。ここでは、少なくとも凸 2 次目的関数を最小化する場合には、これは実際に遭遇しうる問題のタイプをかなりよく代表していることを見る。

2 次凸関数

次の関数を考える。

$$h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{x}^\top \mathbf{c} + b. \tag{15.6.6}$$

これは一般的な 2 次関数である。正定値行列 $\mathbf{Q} \succ 0$、すなわち正の固有値を持つ行列に対しては、これは $\mathbf{x}^* = -\mathbf{Q}^{-1}\mathbf{c}$ で最小値 $b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$ をとる。したがって、h は次のように書き換えられる。

$$h(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})^\top \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c}) + b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}. \tag{15.6.7}$$

勾配は $\partial_{\mathbf{x}} h(\mathbf{x}) = \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$ で与えられる。つまり、$\mathbf{x}$ と最小化点の距離に $\mathbf{Q}$ を掛けたものになっている。したがって、速度も $\mathbf{Q}(\mathbf{x}_t - \mathbf{Q}^{-1}\mathbf{c})$ の項の線形結合になる。

$\mathbf{Q}$ は正定値なので、$\mathbf{Q} = \mathbf{O}^\top \mathbf{\Lambda} \mathbf{O}$ として固有系に分解できる。ここで $\mathbf{O}$ は直交（回転）行列、$\mathbf{\Lambda}$ は正の固有値からなる対角行列である。これにより、$\mathbf{x}$ から $\mathbf{z} \stackrel{\text{def}}{=} \mathbf{O}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$ への変数変換を行うと、はるかに簡単な式が得られる。

$$h(\mathbf{z}) = \frac{1}{2}\mathbf{z}^\top \mathbf{\Lambda} \mathbf{z} + b'. \tag{15.6.8}$$

ここで $b' = b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$ である。$\mathbf{O}$ は単なる直交行列なので、勾配を意味のあ

る形では変化させない。$\mathbf{z}$ で表すと、勾配降下法は

$$\mathbf{z}_t = \mathbf{z}_{t-1} - \boldsymbol{\Lambda}\mathbf{z}_{t-1} = (\mathbf{I} - \boldsymbol{\Lambda})\mathbf{z}_{t-1}. \tag{15.6.9}$$

この式で重要なのは、勾配降下法が異なる固有空間の間で混ざらないことである。つまり、$\mathbf{Q}$ の固有系で表すと、最適化問題は座標ごとに進む。同様のことは次についても成り立つ。

$$\begin{aligned}\mathbf{v}_t &= \beta\mathbf{v}_{t-1} + \boldsymbol{\Lambda}\mathbf{z}_{t-1} \\ \mathbf{z}_t &= \mathbf{z}_{t-1} - \eta\left(\beta\mathbf{v}_{t-1} + \boldsymbol{\Lambda}\mathbf{z}_{t-1}\right) \\ &= (\mathbf{I} - \eta\boldsymbol{\Lambda})\mathbf{z}_{t-1} - \eta\beta\mathbf{v}_{t-1}.\end{aligned} \tag{15.6.10}$$

これにより、次の定理を証明したことになる。凸 2 次関数に対するモメンタムあり・なしの勾配降下法は、2 次行列の固有ベクトル方向に沿った座標ごとの最適化に分解される。

スカラー関数

上の結果を踏まえて、関数 $f(x) = \frac{\lambda}{2}x^2$ を最小化するときに何が起こるか見てみよう。勾配降下法では

$$x_{t+1} = x_t - \eta\lambda x_t = (1 - \eta\lambda)x_t. \tag{15.6.11}$$

$|1 - \eta\lambda| < 1$ である限り、この最適化は指数的な速度で収束する。なぜなら、t ステップ後には $x_t = (1 - \eta\lambda)^t x_0$ となるからである。これは、$\eta\lambda = 1$ までは学習率 η を増やすと収束率が最初は改善することを示している。それを超えると発散し、$\eta\lambda > 2$ では最適化問題は発散する。

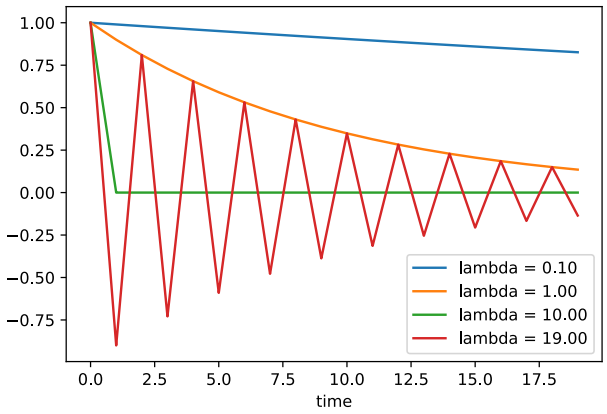
```
lambdas = [0.1, 1, 10, 19]
eta = 0.1
d2l.set_figsize((6, 4))
for lam in lambdas:
    t = d2l.numpy(d2l.arange(20))
    d2l.plt.plot(t, (1 - eta * lam) ** t, label=f'lambda = {lam:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```

モメンタムの場合の収束を解析するために、更新式を 2 つのスカラー、すなわち x 用と速度 v 用に書き直す。すると次が得られる。

$$\begin{bmatrix} v_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \beta & \lambda \\ -\eta\beta & (1 - \eta\lambda) \end{bmatrix} \begin{bmatrix} v_t \\ x_t \end{bmatrix} = \mathbf{R}(\beta, \eta, \lambda) \begin{bmatrix} v_t \\ x_t \end{bmatrix}. \tag{15.6.12}$$

ここでは、収束挙動を支配する 2×2 行列を $\mathbf{R}$ で表した。t ステップ後、初期値 $[v_0, x_0]$ は $\mathbf{R}(\beta, \eta, \lambda)^t [v_0, x_0]$ になる。したがって、収束速度を決めるのは $\mathbf{R}$ の固有値である。優れたアニメーションと詳細な解析については Goh (2017) の Distill の記事[88] と Flammarion and Bach (2015) を参照されたい。$0 < \eta\lambda < 2 + 2\beta$ で速度が収束することを示せる。これは、勾配降下法の $0 < \eta\lambda < 2$ と比べて、許容されるパラメータ範囲が広いことを意味する。また一般に、β の値が大きいほど望

88 



ましいことも示唆している。さらに詳しい内容にはかなりの技術的説明が必要になるため、興味のある読者には原著を参照することを勧める。

15.6.4 まとめ

- モメンタムは、勾配を過去の勾配の漏れのある平均に置き換える。これにより収束が大幅に加速される。
- ノイズのない勾配降下法にも、（ノイズのある）確率的勾配降下法にも有用である。
- モメンタムは、確率的勾配降下法で起こりやすい最適化過程の停滞を防ぐ。
- 過去のデータを指数的に減衰させて重み付けするため、有効な勾配数は $\frac{1}{1-\beta}$ で与えられる。
- 凸 2 次問題の場合、これを詳細に明示的に解析できる。
- 実装はかなり簡単だが、追加の状態ベクトル（速度 $\mathbf{v}$）を保存する必要がある。

15.6.5 演習

1. モメンタムのハイパーパラメータと学習率の他の組み合わせを使い、異なる実験結果を観察して分析せよ。
2. 複数の固有値を持つ 2 次問題、すなわち $f(x) = \frac{1}{2}\sum_i \lambda_i x_i^2$、たとえば $\lambda_i = 2^{-i}$ に対して、勾配降下法とモメンタムを試せ。初期値 $x_i = 1$ のときに x の値がどのように減少するかをプロットせよ。
3. $h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{x}^\top\mathbf{c} + b$ の最小値と最小化点を導出せよ。
4. モメンタム付き確率的勾配降下法を行うと何が変わるか。モメンタム付きミニバッチ確率的勾配降下法を使うと何が起こるか。パラメータを変えて実験せよ。

15.7 Adagrad

まず、まれにしか現れない特徴をもつ学習問題について考えてみよう。

15.7.1 疎な特徴と学習率

言語モデルを学習していると想像してほしい。十分な精度を得るには、通常、学習を続けるにつれて学習率を下げていく必要があり、その減衰率はたいてい $O(t^{-\frac{1}{2}})$ かそれより遅い程度である。ここで、疎な特徴、すなわちごくまれにしか現れない特徴をもつモデルの学習を考える。これは自然言語ではよくあることで、たとえば *learning* よりも *preconditioning* という単語に出会う確率のほうがずっと低いだろう。しかし、計算広告や個人化された協調フィルタリングなど、他の分野でも同様に一般的である。結局のところ、少数の人にしか関心を持たれないものはたくさんある。

まれな特徴に対応するパラメータは、その特徴が現れたときにしか意味のある更新を受けない。学習率が減少していくと、頻出特徴に対応するパラメータは比較的すぐに最適値へ収束する一方で、まれな特徴については、その最適値を決めるのに十分な頻度で観測される前に学習率が小さくなりすぎてしまう、という状況が起こりえる。言い換えると、学習率の減少が、頻出特徴に対しては遅すぎ、まれな特徴に対しては速すぎるのである。

この問題を緩和する一つの手は、特定の特徴を何回見たかを数え、それを学習率調整のための時計として使うことである。つまり、学習率を $\eta = \frac{\eta_0}{\sqrt{t+c}}$ の形で選ぶ代わりに、$\eta_i = \frac{\eta_0}{\sqrt{s(i,t)+c}}$ を使うことができる。ここで $s(i,t)$ は、時刻 t までに観測した特徴 i の非ゼロ回数を数える。これは実際、ほとんど追加コストなしでかなり簡単に実装できる。しかし、データが厳密に疎である場合ではなく、勾配がしばしば非常に小さく、まれに大きくなるようなデータではうまくいかない。そもそも、何を「観測された特徴」とみなすかの境界をどこに引くべきかが明確ではないからである。

Duchi *et al.* (2011) による Adagrad は、このかなり粗いカウンター $s(i,t)$ を、これまでに観測された勾配の二乗の集計に置き換えることでこの問題に対処する。特に、学習率を調整する手段として $s(i,t+1) = s(i,t) + (\partial_i f(\mathbf{x}))^2$ を用いる。これには二つの利点がある。第一に、勾配が十分に大きいかどうかを判断する必要がなくなる。第二に、勾配の大きさに応じて自動的にスケールが調整される。大きな勾配に頻繁に対応する座標は大きく縮小され、一方で小さな勾配しか持たない座標はより穏やかに扱われる。実際には、これにより計算広告や関連問題に対して非常に有効な最適化手法になる。しかし、ここには Adagrad に本来備わっている追加の利点が隠れており、それは前処理の文脈で理解するのが最もよいだろう。

15.7.2 前処理

凸最適化問題は、アルゴリズムの特性を分析するのに適している。というのも、ほとんどの非凸問題では意味のある理論保証を導くのが難しい一方で、直感や 洞察はしばしばそのまま役立つからである。$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{c}^\top\mathbf{x} + b$ を最小化する問題を見てみよう。

Section 15.6 で見たように、この問題は固有値分解 $\mathbf{Q} = \mathbf{U}^\top\mathbf{\Lambda}\mathbf{U}$ を用いて書き換えることができ、各座標を個別に解ける、はるかに単純な問題になる。

$$f(\mathbf{x}) = \bar{f}(\bar{\mathbf{x}}) = \frac{1}{2}\bar{\mathbf{x}}^\top\mathbf{\Lambda}\bar{\mathbf{x}} + \bar{\mathbf{c}}^\top\bar{\mathbf{x}} + b. \tag{15.7.1}$$

ここでは $\bar{\mathbf{x}} = \mathbf{U}\mathbf{x}$、したがって $\bar{\mathbf{c}} = \mathbf{U}\mathbf{c}$ を用いた。変換後の問題の最小解は $\bar{\mathbf{x}} = -\mathbf{\Lambda}^{-1}\bar{\mathbf{c}}$、最小値は $-\frac{1}{2}\bar{\mathbf{c}}^\top\mathbf{\Lambda}^{-1}\bar{\mathbf{c}} + b$ である。$\mathbf{\Lambda}$ は $\mathbf{Q}$ の固有値を並べた対角行列なので、これはずっと計算しやすくなる。

$\mathbf{c}$ を少し摂動したとき、f の最小解も少しだけ変わることを期待したいところである。ところが、実際にはそうならない。$\mathbf{c}$ のわずかな変化は $\bar{\mathbf{c}}$ にも同程度のわずかな変化しか与えないが、f（および $\bar{f}$）の最小解についてはそうではない。固有値 $\mathbf{\Lambda}_i$ が大きいとき、$\bar{x}_i$ と $\bar{f}$ の最小値の変化は小さくなる。逆に、$\mathbf{\Lambda}_i$ が小さいときには $\bar{x}_i$ の変化は劇的になりえる。最大固有値と最小固有値の比は、最適化問題の条件数と呼ばれる。

$$\kappa = \frac{\mathbf{\Lambda}_1}{\mathbf{\Lambda}_d}. \tag{15.7.2}$$

条件数 κ が大きいと、最適化問題を正確に解くのは困難である。値のダイナミックレンジが大きい場合にも正しく扱えるよう、注意深くする必要がある。この分析から、素朴ではあるものの明らかな疑問が生じる。すべての固有値が 1 になるように空間を歪めてしまえば、問題を単純に「修正」できないのだろうか。理論上はこれはかなり簡単である。$\mathbf{Q}$ の固有値と固有ベクトルさえあれば、問題を $\mathbf{x}$ から $\mathbf{z} \stackrel{\text{def}}{=} \mathbf{\Lambda}^{\frac{1}{2}}\mathbf{U}\mathbf{x}$ へと再スケーリングできる。新しい座標系では $\mathbf{x}^\top\mathbf{Q}\mathbf{x}$ は $\|\mathbf{z}\|^2$ に簡約できる。とはいえ、これはかなり実用的でない提案である。一般に、固有値と固有ベクトルを計算することは、実際の問題を解くことよりも はるかに高コストだからである。

固有値を正確に計算するのは高価かもしれないが、それを推測し、多少なりとも近似的に計算するだけでも、何もしないよりはずっとよい場合がある。特に、$\mathbf{Q}$ の対角成分を使って、それに応じて再スケーリングすることができる。これは固有値を計算するよりも はるかに安価である。

$$\tilde{\mathbf{Q}} = \text{diag}^{-\frac{1}{2}}(\mathbf{Q})\mathbf{Q}\text{diag}^{-\frac{1}{2}}(\mathbf{Q}). \tag{15.7.3}$$

このとき $\tilde{\mathbf{Q}}_{ij} = \mathbf{Q}_{ij}/\sqrt{\mathbf{Q}_{ii}\mathbf{Q}_{jj}}$ であり、特にすべての i について $\tilde{\mathbf{Q}}_{ii} = 1$ である。多くの場合、これにより条件数はかなり改善される。たとえば、先ほど議論したケースでは、問題が軸に整列しているため、この処理だけで問題を完全に解消できる。

残念ながら、ここでも別の問題に直面する。深層学習では通常、目的関数の二階微分にすらアクセスできない。$\mathbf{x} \in \mathbb{R}^d$ に対して、ミニバッチ上であっても二階微分の計算には $O(d^2)$ の空間と計算量が必要になることがあり、実用的では

ない。Adagrad の巧妙なアイデアは、この捉えにくいヘッセ行列の対角成分の代わりに、比較的安価に計算でき、しかも有効な代理量、すなわち勾配そのものの大きさを使うことである。

なぜこれがうまくいくのかを見るために、$\bar{f}(\bar{\mathbf{x}})$ を考えよう。すると

$$\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}}) = \boldsymbol{\Lambda} \bar{\mathbf{x}} + \bar{\mathbf{c}} = \boldsymbol{\Lambda} (\bar{\mathbf{x}} - \bar{\mathbf{x}}_0), \tag{15.7.4}$$

ここで $\bar{\mathbf{x}}_0$ は $\bar{f}$ の最小解である。したがって、勾配の大きさは $\boldsymbol{\Lambda}$ と最適解からの距離の両方に依存する。もし $\bar{\mathbf{x}} - \bar{\mathbf{x}}_0$ が変化しないなら、これで十分である。結局のところ、この場合には勾配 $\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}})$ の大きさだけで足りる。AdaGrad は確率的勾配降下法なので、最適点においても分散がゼロでない勾配が現れる。その結果、勾配の分散をヘッセ行列のスケールの安価な代理として安全に使うことができる。厳密な解析はこの節の範囲を超える（数ページにわたる内容になる）。詳細は (Duchi *et al.*, 2011) を参照されたい。

15.7.3 アルゴリズム

上の議論を形式化しよう。過去の勾配分散を次のように蓄積するために、変数 $\mathbf{s}_t$ を用いる。

$$\begin{aligned} \mathbf{g}_t &= \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t. \end{aligned} \tag{15.7.5}$$

ここでの演算は座標ごとに適用される。つまり、$\mathbf{v}^2$ の各成分は v_i^2 である。同様に、$\frac{1}{\sqrt{v}}$ の各成分は $\frac{1}{\sqrt{v_i}}$ であり、$\mathbf{u} \cdot \mathbf{v}$ の各成分は $u_i v_i$ である。これまでと同様に、η は学習率、ϵ は 0 で割らないようにするための加算定数である。最後に、$\mathbf{s}_0 = \mathbf{0}$ で初期化する。

モメンタム法と同様に、補助変数を追跡する必要があるが、今回は各座標ごとに個別の学習率を可能にするためである。主な計算コストは通常 $l(y_t, f(\mathbf{x}_t, \mathbf{w}))$ とその導関数の計算であるため、Adagrad のコストは SGD と比べて大きくは増えない。

$\mathbf{s}_t$ に二乗勾配を蓄積するということは、$\mathbf{s}_t$ が本質的に線形の速度で増加することを意味する（実際には、勾配が最初は減少するので、ややそれより遅い）。これにより、座標ごとに調整された $O(t^{-\frac{1}{2}})$ の学習率が得られる。凸問題に対しては、これはまったく十分である。しかし深層学習では、学習率をもう少しゆっくり下げたい場合がある。そのため、後続の章で議論するいくつかの Adagrad 変種が生まれた。ここではまず、二次の凸問題でどのように振る舞うかを見てみよう。先ほどと同じ問題を使う。

$$f(\mathbf{x}) = 0.1 x_1^2 + 2 x_2^2. \tag{15.7.6}$$

以前と同じ学習率、すなわち $\eta = 0.4$ を用いて Adagrad を実装する。見てわかるように、独立変数の反復軌跡はより滑らかになる。しかし、$\boldsymbol{s}_t$ の累積効果によ

り学習率は継続的に減衰するため、反復の後半では独立変数の動きはそれほど大きくない。

```
%matplotlib inline
from d2l import torch as d2l
import math
import torch
```

```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import np, npx
npx.set_np()
```

```
def adagrad_2d(x1, x2, s1, s2):
    eps = 1e-6
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1 ** 2
    s2 += g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta = 0.4
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

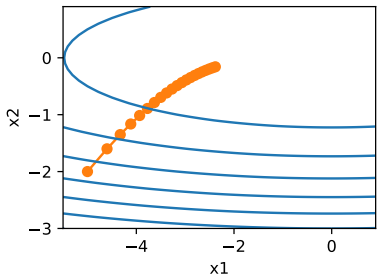
```
%matplotlib inline
from d2l import tensorflow as d2l
import math
import tensorflow as tf
```

```
def adagrad_2d(x1, x2, s1, s2):
    eps = 1e-6
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1 ** 2
    s2 += g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta = 0.4
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
epoch 20, x1: -2.382563, x2: -0.158591
```

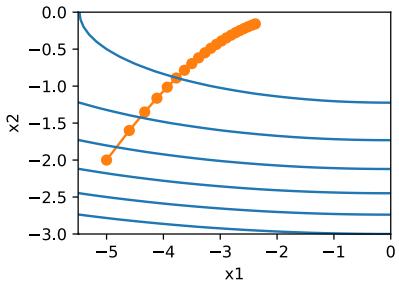



```
def adagrad_2d(x1, x2, s1, s2):
    eps = 1e-6
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1 ** 2
    s2 += g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta = 0.4
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
epoch 20, x1: -2.382563, x2: -0.158591
[07:17:11] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```




```
eta = 2
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
def adagrad_2d(x1, x2, s1, s2):
    eps = 1e-6
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1 ** 2
```

(continues on next page)

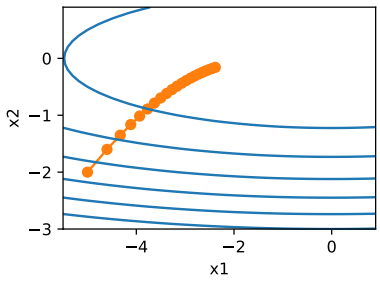
(continued from previous page)

```
    s2 += g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta = 0.4
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

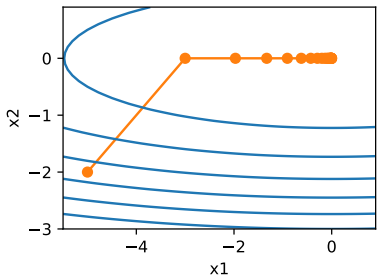
```
epoch 20, x1: -2.382563, x2: -0.158591
```



学習率を 2 に増やすと、よりよい挙動が見られる。これは、ノイズのない場合であっても学習率の減少がかなり急激になりうることを示しており、パラメータが適切に収束するように注意する必要があることがわかる。

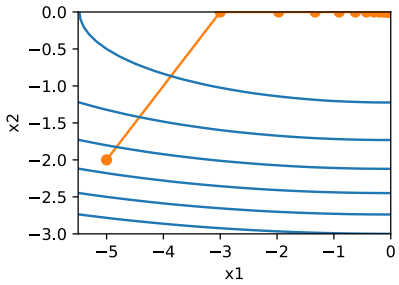
```
eta = 2
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
epoch 20, x1: -0.002295, x2: -0.000000
```




```
eta = 2
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

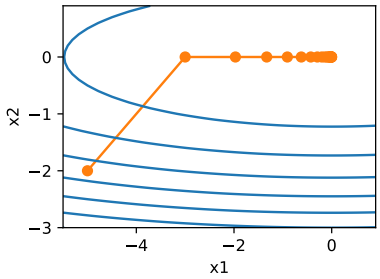
```
epoch 20, x1: -0.002295, x2: -0.000000
```




```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adagrad, init_adagrad_states(feature_dim),
               {'lr': 0.1}, data_iter, feature_dim);
```

```
eta = 2
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
epoch 20, x1: -0.002295, x2: -0.000000
```



15.7.4 ゼロからの実装

モメンタム法と同様に、Adagrad でもパラメータと同じ形状の状態変数を保持する必要がある。

```
def init_adagrad_states(feature_dim):
    s_w = d2l.zeros((feature_dim, 1))
    s_b = d2l.zeros(1)
    return (s_w, s_b)

def adagrad(params, states, hyperparams):
    eps = 1e-6
    for p, s in zip(params, states):
```

(continues on next page)

(continued from previous page)

```
        with torch.no_grad():
            s[:] += torch.square(p.grad)
            p[:] -= hyperparams['lr'] * p.grad / torch.sqrt(s + eps)
        p.grad.data.zero_()
```

```
def init_adagrad_states(feature_dim):
    s_w = d2l.zeros((feature_dim, 1))
    s_b = d2l.zeros(1)
    return (s_w, s_b)

def adagrad(params, states, hyperparams):
    eps = 1e-6
    for p, s in zip(params, states):
        s[:] += np.square(p.grad)
        p[:] -= hyperparams['lr'] * p.grad / np.sqrt(s + eps)
```

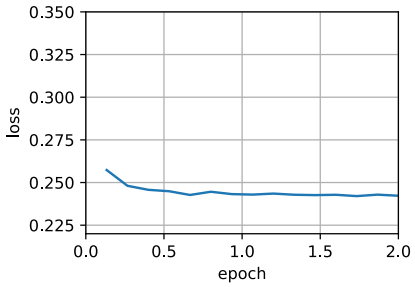
```
def init_adagrad_states(feature_dim):
    s_w = tf.Variable(d2l.zeros((feature_dim, 1)))
    s_b = tf.Variable(d2l.zeros(1))
    return (s_w, s_b)

def adagrad(params, grads, states, hyperparams):
    eps = 1e-6
    for p, s, g in zip(params, states, grads):
        s[:].assign(s + tf.math.square(g))
        p[:].assign(p - hyperparams['lr'] * g / tf.math.sqrt(s + eps))
```

Section 15.5 の実験と比べて、モデルの学習にはより大きな学習率を使う。

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adagrad, init_adagrad_states(feature_dim),
               {'lr': 0.1}, data_iter, feature_dim);
```

```
loss: 0.242, 0.136 sec/epoch
```

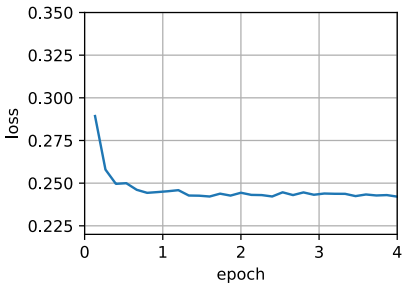


15.7.5 簡潔な実装

アルゴリズム adagrad の Trainer インスタンスを使えば、Gluon で Adagrad アルゴリズムを呼び出せる。

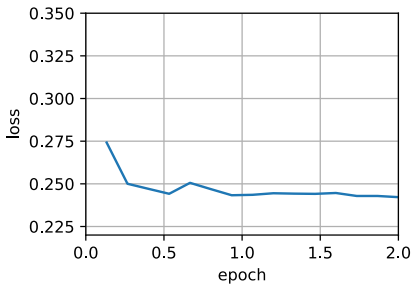
```
trainer = torch.optim.Adagrad
d2l.train_concise_ch11(trainer, {'lr': 0.1}, data_iter)
```

```
loss: 0.242, 0.108 sec/epoch
```




```
d2l.train_concise_ch11('adagrad', {'learning_rate': 0.1}, data_iter)
```

```
loss: 0.242, 1.009 sec/epoch
```

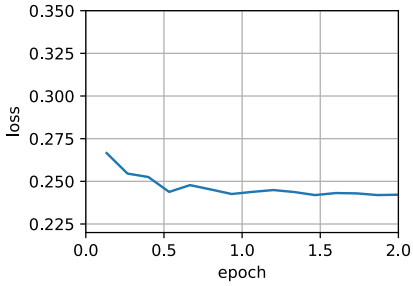



```
trainer = tf.keras.optimizers.Adagrad
d2l.train_concise_ch11(trainer, {'learning_rate' : 0.1}, data_iter)
```

```
loss: 0.242, 1.205 sec/epoch
```

15.7.6 まとめ

- Adagrad は、座標ごとに動的に学習率を下げる。
- 勾配の大きさを、進捗の速さを調整する手段として用いる。勾配が大きい座標は、より小さな学習率で補正される。



- 深層学習の問題では、メモリと計算量の制約のため、正確な二階微分の計算は通常不可能である。勾配は有用な代理量になりえる。
- 最適化問題の構造がかなり不均一な場合、Adagrad はその歪みを緩和するのに役立つ。
- Adagrad は、まれにしか現れない項に対して学習率をよりゆっくり下げる必要がある疎な特徴に特に有効である。
- 深層学習の問題では、Adagrad は学習率を下げるのが強すぎる場合がある。これを緩和する方法については Section 15.10 で議論する。

15.7.7 演習

1. 直交行列 $\mathbf{U}$ とベクトル $\mathbf{c}$ に対して、次が成り立つことを証明せよ: $\|\mathbf{c} - \mathbf{ffi}\|_2 = \|\mathbf{Uc} - \mathbf{Uffi}\|_2$. なぜこれは、直交変数変換の後でも摂動の大きさが変わらないことを意味するのか。
2. $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ に対して、また目的関数を 45 度回転させた $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$ に対して Adagrad を試してみよ。挙動は異なるか。
3. 行列 $\mathbf{M}$ の固有値 λ_i が少なくとも一つの j の選び方について $|\lambda_i - \mathbf{M}_{jj}| \leq \sum_{k \neq j} |\mathbf{M}_{jk}|$ を満たすことを述べる Gerschgorin の円定理[89] を証明せよ。
4. Gerschgorin の定理は、対角前処理された行列 $\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})\mathbf{M}\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})$ の固有値について何を教えてくれるか。
5. Fashion-MNIST に適用した Section 10.6 のような適切な深層ネットワークで Adagrad を試してみよ。
6. 学習率の減衰をより緩やかにするには、Adagrad をどのように修正する必要があるか。

15.8 RMSProp

Section 15.7 における重要な問題の 1 つは、学習率が事実上 $O(t^{-\frac{1}{2}})$ のあらかじめ定められたスケジュールで減少していくことである。これは一般に凸問題には適しているが、深層学習で遭遇するような非凸問題には必ずしも理想的ではない。それでも、Adagrad の座標ごとの適応性は、前処理器として非常に望ましいものである。

Tieleman and Hinton (2012) は、スケジュールによる学習率の調整と座標適応的な学習率を切り離すための簡単な修正として、RMSProp アルゴリズムを提案した。問題は、Adagrad が勾配 $\mathbf{g}_t$ の二乗を状態ベクトル $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$ に蓄積していくことである。その結果、正規化がないため $\mathbf{s}_t$ は際限なく増大し続け、アルゴリズムが収束するにつれて本質的には線形に増えていく。

この問題を修正する 1 つの方法は、$\mathbf{s}_t / t$ を使うことである。$\mathbf{g}_t$ の分布が適切であれば、これは収束する。残念ながら、この手続きは値の完全な軌跡を記憶するため、極限的な挙動が重要になるまでに非常に長い時間がかかるかもしれない。別の方法として、モメンタム法で用いたのと同じようにリーキー平均を使うことができる。すなわち、あるパラメータ $\gamma > 0$ に対して $\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2$ とする。他の部分をすべて変えずに残すと RMSProp になる。

15.8.1 アルゴリズム

式を詳しく書き下してみよう。

$$\begin{aligned} \mathbf{s}_t &\leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t. \end{aligned} \tag{15.8.1}$$

定数 $\epsilon > 0$ は通常 10^{-6} に設定され、ゼロ除算や過度に大きなステップサイズを避けるために用いられる。この展開により、学習率 η を、各座標ごとに適用されるスケーリングとは独立に制御できるようになる。リーキー平均の観点では、これまでモメンタム法で行ったのと同じ推論を適用できる。$\mathbf{s}_t$ の定義を展開すると

$$\begin{aligned} \mathbf{s}_t &= (1-\gamma)\mathbf{g}_t^2 + \gamma \mathbf{s}_{t-1} \\ &= (1-\gamma)\left(\mathbf{g}_t^2 + \gamma \mathbf{g}_{t-1}^2 + \gamma^2 \mathbf{g}_{t-2} + \ldots,\right). \end{aligned} \tag{15.8.2}$$

Section 15.6 と同様に、$1 + \gamma + \gamma^2 + \ldots, = \frac{1}{1-\gamma}$ を用いる。したがって、重みの総和は 1 に正規化され、観測の半減期は γ^{-1} になる。さまざまな γ の選択について、過去 40 ステップの重みを可視化してみよう。

```
from d2l import torch as d2l
import torch
import math
```

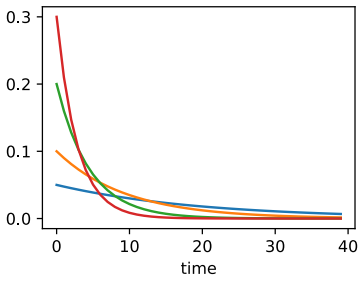
```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import np, npx

npx.set_np()
```

```
d2l.set_figsize()
gammas = [0.95, 0.9, 0.8, 0.7]
for gamma in gammas:
    x = d2l.numpy(d2l.arange(40))
    d2l.plt.plot(x, (1-gamma) * gamma ** x, label=f'gamma = {gamma:.2f}')
d2l.plt.xlabel('time');
```

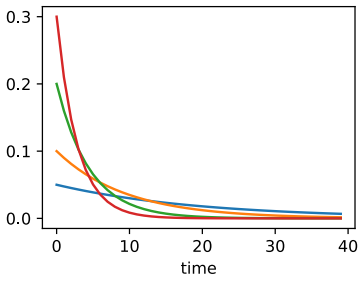
```
from d2l import tensorflow as d2l
import tensorflow as tf
import math
```

```
d2l.set_figsize()
gammas = [0.95, 0.9, 0.8, 0.7]
for gamma in gammas:
    x = d2l.numpy(d2l.arange(40))
    d2l.plt.plot(x, (1-gamma) * gamma ** x, label=f'gamma = {gamma:.2f}')
d2l.plt.xlabel('time');
```




```
d2l.set_figsize()
gammas = [0.95, 0.9, 0.8, 0.7]
for gamma in gammas:
    x = d2l.numpy(d2l.arange(40))
    d2l.plt.plot(x, (1-gamma) * gamma ** x, label=f'gamma = {gamma:.2f}')
d2l.plt.xlabel('time');
```

```
[06:58:30] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

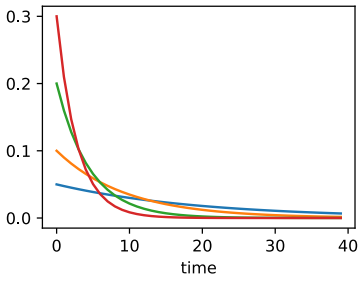



```
def rmsprop_2d(x1, x2, s1, s2):
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
    s1 = gamma * s1 + (1 - gamma) * g1 ** 2
    s2 = gamma * s2 + (1 - gamma) * g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta, gamma = 0.4, 0.9
d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))
```

```
d2l.set_figsize()
gammas = [0.95, 0.9, 0.8, 0.7]
for gamma in gammas:
    x = d2l.numpy(d2l.arange(40))
    d2l.plt.plot(x, (1-gamma) * gamma ** x, label=f'gamma = {gamma:.2f}')
d2l.plt.xlabel('time');
```



15.8.2 ゼロからの実装

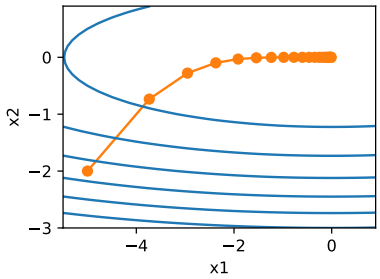
これまでと同様に、二次関数 $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ を用いて RMSProp の軌跡を観察する。Section 15.7 では、学習率を 0.4 にして Adagrad を使ったとき、学習率が急速に下がりすぎるため、アルゴリズムの後半で変数の移動が非常に遅くなったことを思い出してほしい。η は別々に制御されるので、RMSProp ではこのようなことは起こらない。

```
def rmsprop_2d(x1, x2, s1, s2):
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
    s1 = gamma * s1 + (1 - gamma) * g1 ** 2
    s2 = gamma * s2 + (1 - gamma) * g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta, gamma = 0.4, 0.9
d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))
```

```
epoch 20, x1: -0.010599, x2: 0.000000
```




```
def rmsprop_2d(x1, x2, s1, s2):
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
    s1 = gamma * s1 + (1 - gamma) * g1 ** 2
    s2 = gamma * s2 + (1 - gamma) * g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

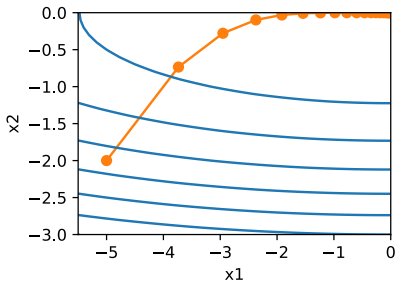
eta, gamma = 0.4, 0.9
d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))
```

```
epoch 20, x1: -0.010599, x2: 0.000000
```

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(rmsprop, init_rmsprop_states(feature_dim),
               {'lr': 0.01, 'gamma': 0.9}, data_iter, feature_dim);
```

```
def rmsprop_2d(x1, x2, s1, s2):
```

(continues on next page)



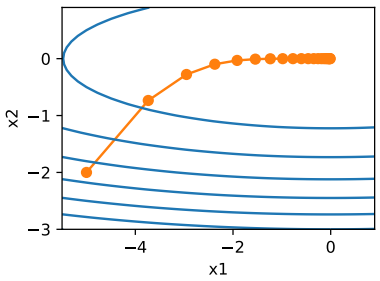
(continued from previous page)

```
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
    s1 = gamma * s1 + (1 - gamma) * g1 ** 2
    s2 = gamma * s2 + (1 - gamma) * g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta, gamma = 0.4, 0.9
d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))
```

```
epoch 20, x1: -0.010599, x2: 0.000000
```



次に、深層ネットワークで使うための RMSProp を実装する。こちらも同様に簡単である。

```
def init_rmsprop_states(feature_dim):
    s_w = d2l.zeros((feature_dim, 1))
    s_b = d2l.zeros(1)
    return (s_w, s_b)
```

```
def init_rmsprop_states(feature_dim):
    s_w = d2l.zeros((feature_dim, 1))
```

(continues on next page)

(continued from previous page)

```
    s_b = d2l.zeros(1)
    return (s_w, s_b)
```

```
def init_rmsprop_states(feature_dim):
    s_w = tf.Variable(d2l.zeros((feature_dim, 1)))
    s_b = tf.Variable(d2l.zeros(1))
    return (s_w, s_b)
```

```
def rmsprop(params, states, hyperparams):
    gamma, eps = hyperparams['gamma'], 1e-6
    for p, s in zip(params, states):
        with torch.no_grad():
            s[:] = gamma * s + (1 - gamma) * torch.square(p.grad)
            p[:] -= hyperparams['lr'] * p.grad / torch.sqrt(s + eps)
        p.grad.data.zero_()
```

```
def rmsprop(params, states, hyperparams):
    gamma, eps = hyperparams['gamma'], 1e-6
    for p, s in zip(params, states):
        s[:] = gamma * s + (1 - gamma) * np.square(p.grad)
        p[:] -= hyperparams['lr'] * p.grad / np.sqrt(s + eps)
```

```
def rmsprop(params, grads, states, hyperparams):
    gamma, eps = hyperparams['gamma'], 1e-6
    for p, s, g in zip(params, states, grads):
        s[:].assign(gamma * s + (1 - gamma) * tf.math.square(g))
        p[:].assign(p - hyperparams['lr'] * g / tf.math.sqrt(s + eps))
```

初期学習率を 0.01、重み付け項 γ を 0.9 に設定する。つまり、s は平均して過去 $1/(1-\gamma) = 10$ 個の二乗勾配を集約する。

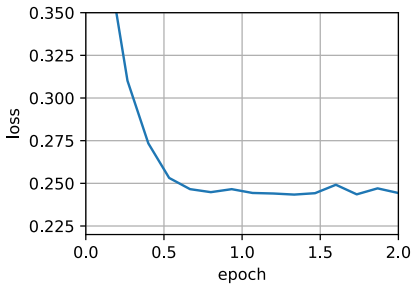
```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(rmsprop, init_rmsprop_states(feature_dim),
               {'lr': 0.01, 'gamma': 0.9}, data_iter, feature_dim);
```

```
loss: 0.244, 0.246 sec/epoch
```

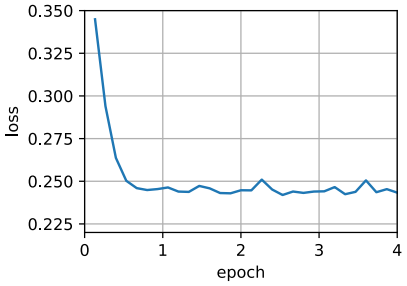
15.8.3 簡潔な実装

RMSProp はかなり人気のあるアルゴリズムなので、Trainer インスタンスでも利用できる。必要なのは、rmsprop という名前のアルゴリズムを使ってインスタンス化し、γ をパラメータ gamma1 に割り当てることだけである。

```
trainer = torch.optim.RMSprop
d2l.train_concise_ch11(trainer, {'lr': 0.01, 'alpha': 0.9},
                       data_iter)
```

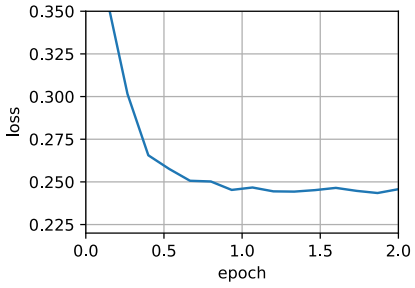



```
loss: 0.243, 0.133 sec/epoch
```



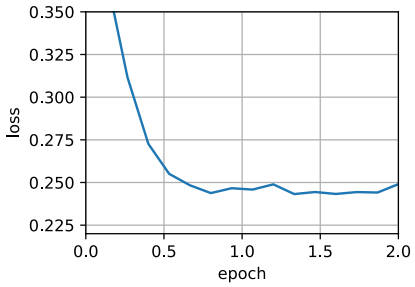

```
d2l.train_concise_ch11('rmsprop', {'learning_rate': 0.01, 'gamma1': 0.9},
                       data_iter)
```

```
loss: 0.246, 0.355 sec/epoch
```




```
trainer = tf.keras.optimizers.RMSprop
d2l.train_concise_ch11(trainer, {'learning_rate': 0.01, 'rho': 0.9},
                       data_iter)
```

```
loss: 0.249, 1.316 sec/epoch
```



15.8.4 まとめ

- RMSProp は Adagrad と非常によく似ており、どちらも勾配の二乗を用いて係数をスケーリングする。
- RMSProp はモメンタムとリーキー平均を共有している。ただし、RMSProp ではこの手法を係数ごとの前処理器の調整に用いる。
- 実際には、学習率は実験者がスケジュール調整する必要がある。
- 係数 γ は、各座標のスケールを調整するときにどれだけ長い履歴を考慮するかを決定する。

15.8.5 演習

1. $\gamma = 1$ とすると、実験的に何が起こるか。なぜか。
2. 最適化問題を回転させて、$f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$ を最小化してみよ。収束はどうなるか。
3. Fashion-MNIST での学習など、実際の機械学習問題で RMSProp を試してみよ。学習率の調整方法を変えて実験せよ。
4. 最適化が進むにつれて γ を調整したいか。RMSProp はこれにどれくらい敏感か。

15.9 Adadelta

Adadelta は AdaGrad (Section 15.7) のもう一つの変種である。主な違いは、学習率が座標ごとに適応的に変化する度合いを小さくしている点にある。さらに、伝統的には、変化量そのものを将来の変化の較正に用いるため、学習率を持たない手法だとされている。このアルゴリズムは Zeiler (2012) で提案された。こ

こまでの前節までのアルゴリズムの議論を踏まえると、かなり素直に理解できる。

15.9.1 アルゴリズム

要するに、Adadelta は 2 つの状態変数を使う。$\mathbf{s}_t$ は勾配の 2 次モーメントのリーキー平均を保持し、$\Delta\mathbf{x}_t$ はモデル自体のパラメータ変化の 2 次モーメントのリーキー平均を保持する。なお、他の文献や実装との互換性のため、著者らの元の記法と命名をそのまま用いている（モメンタム、Adagrad、RMSProp、Adadelta で同じ目的の変数を表すのに、わざわざ異なるギリシャ文字を使うべき実質的な理由はない）。

以下に Adadelta の技術的詳細を示す。ここでのパラメータを ρ とすると、Section 15.8 と同様に次のリーキー更新を得る。

$$\mathbf{s}_t = \rho \mathbf{s}_{t-1} + (1-\rho)\mathbf{g}_t^2. \tag{15.9.1}$$

Section 15.8 との違いは、再スケールした勾配 $\mathbf{g}_t'$ を用いて更新を行う点である。すなわち、

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \mathbf{g}_t'. \tag{15.9.2}$$

では、再スケールした勾配 $\mathbf{g}_t'$ とは何だろうか。次のように計算できる。

$$\mathbf{g}_t' = \frac{\sqrt{\Delta\mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t, \tag{15.9.3}$$

ここで $\Delta\mathbf{x}_{t-1}$ は、再スケールした勾配 $\mathbf{g}_t'$ の二乗のリーキー平均である。$\Delta\mathbf{x}_0$ を 0 に初期化し、各ステップで $\mathbf{g}_t'$ を用いて更新する。すなわち、

$$\Delta\mathbf{x}_t = \rho\Delta\mathbf{x}_{t-1} + (1-\rho)\mathbf{g}_t'^2, \tag{15.9.4}$$

また、数値安定性を保つために ϵ（10^{-5} のような小さな値）を加える。

15.9.2 実装

Adadelta では、各変数ごとに 2 つの状態変数 $\mathbf{s}_t$ と $\Delta\mathbf{x}_t$ を保持する必要がある。これにより、次の実装になる。

```
%matplotlib inline
from d2l import torch as d2l
import torch

def init_adadelta_states(feature_dim):
    s_w, s_b = d2l.zeros((feature_dim, 1)), d2l.zeros(1)
    delta_w, delta_b = d2l.zeros((feature_dim, 1)), d2l.zeros(1)
    return ((s_w, delta_w), (s_b, delta_b))

def adadelta(params, states, hyperparams):
    rho, eps = hyperparams['rho'], 1e-5
    for p, (s, delta) in zip(params, states):
```

(continues on next page)

(continued from previous page)

```
        with torch.no_grad():
            # In-place updates via [:]
            s[:] = rho * s + (1 - rho) * torch.square(p.grad)
            g = (torch.sqrt(delta + eps) / torch.sqrt(s + eps)) * p.grad
            p[:] -= g
            delta[:] = rho * delta + (1 - rho) * g * g
        p.grad.data.zero_()
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()

def init_adadelta_states(feature_dim):
    s_w, s_b = d2l.zeros((feature_dim, 1)), d2l.zeros(1)
    delta_w, delta_b = d2l.zeros((feature_dim, 1)), d2l.zeros(1)
    return ((s_w, delta_w), (s_b, delta_b))

def adadelta(params, states, hyperparams):
    rho, eps = hyperparams['rho'], 1e-5
    for p, (s, delta) in zip(params, states):
        # In-place updates via [:]
        s[:] = rho * s + (1 - rho) * np.square(p.grad)
        g = (np.sqrt(delta + eps) / np.sqrt(s + eps)) * p.grad
        p[:] -= g
        delta[:] = rho * delta + (1 - rho) * g * g
```

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adadelta, init_adadelta_states(feature_dim),
               {'rho': 0.9}, data_iter, feature_dim);
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf

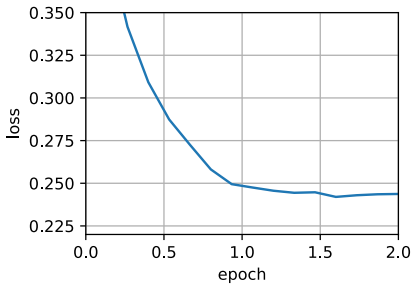
def init_adadelta_states(feature_dim):
    s_w = tf.Variable(d2l.zeros((feature_dim, 1)))
    s_b = tf.Variable(d2l.zeros(1))
    delta_w = tf.Variable(d2l.zeros((feature_dim, 1)))
    delta_b = tf.Variable(d2l.zeros(1))
    return ((s_w, delta_w), (s_b, delta_b))

def adadelta(params, grads, states, hyperparams):
    rho, eps = hyperparams['rho'], 1e-5
    for p, (s, delta), grad in zip(params, states, grads):
        s[:].assign(rho * s + (1 - rho) * tf.math.square(grad))
        g = (tf.math.sqrt(delta + eps) / tf.math.sqrt(s + eps)) * grad
        p[:].assign(p - g)
        delta[:].assign(rho * delta + (1 - rho) * g * g)
```

$\rho = 0.9$ を選ぶと、各パラメータ更新の半減期は 10 になる。これはかなりうまく機能する傾向がある。結果は次のようになる。

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adadelta, init_adadelta_states(feature_dim),
               {'rho': 0.9}, data_iter, feature_dim);
```

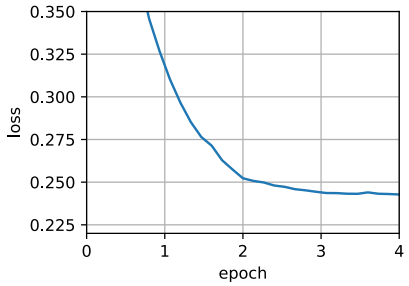
```
loss: 0.244, 0.151 sec/epoch
```



簡潔な実装としては、高レベル API の Adadelta アルゴリズムをそのまま使う。これにより、より簡潔な呼び出しを 1 行で書ける。

```
trainer = torch.optim.Adadelta
d2l.train_concise_ch11(trainer, {'rho': 0.9}, data_iter)
```

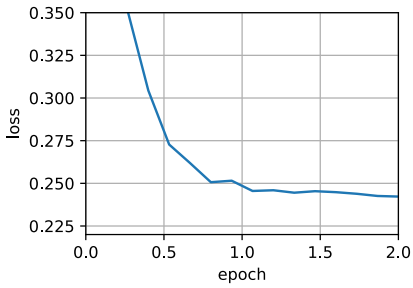
```
loss: 0.243, 0.121 sec/epoch
```



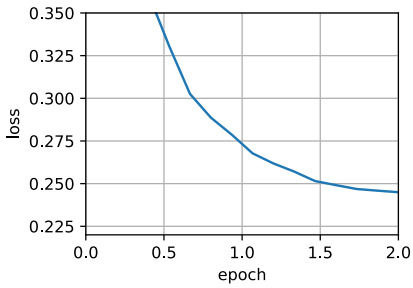

```
d2l.train_concise_ch11('adadelta', {'rho': 0.9}, data_iter)
```

```
loss: 0.242, 0.789 sec/epoch
```

```
# adadelta is not converging at default learning rate
# but it is converging at lr = 5.0
trainer = tf.keras.optimizers.Adadelta
d2l.train_concise_ch11(trainer, {'learning_rate':5.0, 'rho': 0.9}, data_iter)
```




```
loss: 0.245, 1.584 sec/epoch
```



15.9.3 まとめ

- Adadelta には学習率パラメータがない。その代わりに、パラメータの変化率そのものを用いて学習率を適応させる。
- Adadelta では、勾配とパラメータ変化の 2 次モーメントを保存するために 2 つの状態変数が必要である。
- Adadelta はリーキー平均を用いて、適切な統計量の逐次推定を維持する。

15.9.4 演習

1. ρ の値を調整してみよ。何が起こるか。
2. $\mathbf{g}_t'$ を使わずにアルゴリズムを実装する方法を示せ。なぜそれがよい考えなのか。
3. Adadelta は本当に学習率不要なのか。Adadelta を破綻させる最適化問題を見つけられるか。
4. Adadelta と Adagrad、RMS prop を比較し、それらの収束挙動について議論せよ。

15.10 Adam

この節に入るまでの議論では、効率的な最適化のためのいくつかの手法に出会った。ここでそれらを詳しく振り返っておこう。

- Section 15.4 で、たとえば冗長なデータに対する本質的な頑健性のおかげで、最適化問題を解く際には勾配降下法よりも確率的勾配降下法のほうが有効であることを見た。
- Section 15.5 で、1 つのミニバッチにより多くの観測を用いるベクトル化によって生じる、さらなる大きな効率向上を見た。これは、効率的なマルチマシン、マルチ GPU、そして全体としての並列処理の鍵である。
- Section 15.6 では、過去の勾配の履歴を集約して収束を加速する仕組みを追加した。
- Section 15.7 では、座標ごとのスケーリングを用いて、計算効率のよい前処理器を実現した。
- Section 15.8 では、座標ごとのスケーリングを学習率調整から切り離した。

Adam (Kingma and Ba, 2014) は、これらすべての手法を 1 つの効率的な学習アルゴリズムにまとめたものである。予想どおり、これは深層学習で用いる最適化アルゴリズムの中でも、より頑健で有効なものとしてかなり人気のあるアルゴリズムになった。ただし、問題がないわけではない。特に (Reddi *et al.*, 2019) は、分散の制御が不十分なために Adam が発散しうる状況があることを示している。続く研究で Zaheer *et al.* (2018) は、これらの問題に対処する Adam の修正版である Yogi を提案した。これについては後で詳しく述べる。ここではまず Adam アルゴリズムを見ていこう。

15.10.1 アルゴリズム

Adam の重要な構成要素の 1 つは、指数加重移動平均（指数移動平均、あるいは leaky averaging とも呼ばれる）を用いて、モーメンタムと勾配の 2 次モーメントの両方の推定値を得ることである。つまり、次の状態変数を用いる。

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)\mathbf{g}_t,\\ \mathbf{s}_t &\leftarrow \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2.\end{aligned} \tag{15.10.1}$$

ここで β_1 と β_2 は非負の重み付けパラメータである。一般的な値としては $\beta_1 = 0.9$ と $\beta_2 = 0.999$ がよく使われる。つまり、分散の推定値はモーメンタム項よりも はるかにゆっくりと変化する。$\mathbf{v}_0 = \mathbf{s}_0 = 0$ と初期化すると、最初のうちは小さい値にかなり強く偏ることに注意されたい。これは、$\sum_{i=0}^{t-1} \beta^i = \frac{1-\beta^t}{1-\beta}$ という事実を用いて項を再正規化することで対処できる。対応する正規化済みの状態変数は

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2^t}. \tag{15.10.2}$$

適切な推定値が得られたので、更新式を書けるようになった。まず、RMSPropと非常によく似た方法で勾配を再スケーリングして、次を得る。

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}. \tag{15.10.3}$$

RMSPropと異なり、更新には勾配そのものではなくモーメンタム $\hat{\mathbf{v}}_t$ を使う。さらに、再スケーリングが $\frac{1}{\sqrt{\hat{\mathbf{s}}_t}+\epsilon}$ を用いて行われる点で、見た目に少し違いがある。前者のほうが実践上わずかにうまくいくと考えられているため、RMSPropからこのように少し変えている。通常は、数値安定性と忠実性のよいトレードオフを得るために $\epsilon = 10^{-6}$ を選ぶ。

これで更新を計算するための材料はすべてそろった。やや拍子抜けするほど単純で、更新は次の形になる。

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}_t'. \tag{15.10.4}$$

Adamの設計を見直すと、その着想は明らかである。モーメンタムとスケールが状態変数の中にはっきり現れている。かなり独特な定義のため、項のバイアス補正が必要になる（これは、初期化と更新条件を少し変えれば修正できる）。第2に、RMSPropがあるので、2つの項の組み合わせはかなり直接的である。最後に、明示的な学習率 η により、収束の問題に対処するためのステップ長を制御できる。

15.10.2 実装

Adamをゼロから実装するのはそれほど難しくない。便宜上、時刻ステップのカウンタ t を `hyperparams` 辞書に保存する。それ以外はすべて単純である。

```
%matplotlib inline
from d2l import torch as d2l
import torch

def init_adam_states(feature_dim):
    v_w, v_b = d2l.zeros((feature_dim, 1)), d2l.zeros(1)
    s_w, s_b = d2l.zeros((feature_dim, 1)), d2l.zeros(1)
    return ((v_w, s_w), (v_b, s_b))

def adam(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s) in zip(params, states):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
            s[:] = beta2 * s + (1 - beta2) * torch.square(p.grad)
            v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
            s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
            p[:] -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr)
                                                       + eps)
        p.grad.data.zero_()
    hyperparams['t'] += 1
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()

def init_adam_states(feature_dim):
    v_w, v_b = d2l.zeros((feature_dim, 1)), d2l.zeros(1)
    s_w, s_b = d2l.zeros((feature_dim, 1)), d2l.zeros(1)
    return ((v_w, s_w), (v_b, s_b))

def adam(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s) in zip(params, states):
        v[:] = beta1 * v + (1 - beta1) * p.grad
        s[:] = beta2 * s + (1 - beta2) * np.square(p.grad)
        v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
        s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
        p[:] -= hyperparams['lr'] * v_bias_corr / (np.sqrt(s_bias_corr) + eps)
    hyperparams['t'] += 1
```

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adam, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf

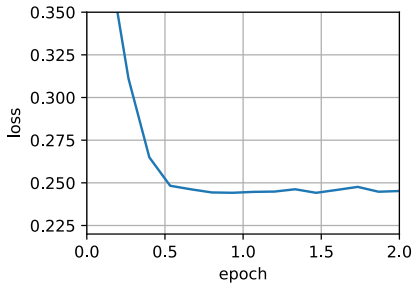
def init_adam_states(feature_dim):
    v_w = tf.Variable(d2l.zeros((feature_dim, 1)))
    v_b = tf.Variable(d2l.zeros(1))
    s_w = tf.Variable(d2l.zeros((feature_dim, 1)))
    s_b = tf.Variable(d2l.zeros(1))
    return ((v_w, s_w), (v_b, s_b))

def adam(params, grads, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s), grad in zip(params, states, grads):
        v[:].assign(beta1 * v  + (1 - beta1) * grad)
        s[:].assign(beta2 * s + (1 - beta2) * tf.math.square(grad))
        v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
        s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
        p[:].assign(p - hyperparams['lr'] * v_bias_corr
                    / tf.math.sqrt(s_bias_corr) + eps)
```

Adam を使ってモデルを学習する準備が整った。学習率は $\eta = 0.01$ を使う。

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adam, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

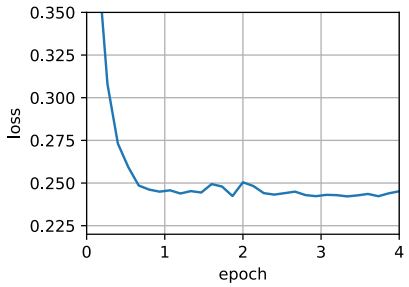
```
loss: 0.245, 0.172 sec/epoch
```



adam は Gluon の trainer 最適化ライブラリの一部として提供されているアルゴリズムの 1 つなので、より簡潔な実装も容易である。したがって、Gluon での実装に必要なのは設定パラメータを渡すことだけである。

```
trainer = torch.optim.Adam
d2l.train_concise_ch11(trainer, {'lr': 0.01}, data_iter)
```

```
loss: 0.245, 0.128 sec/epoch
```



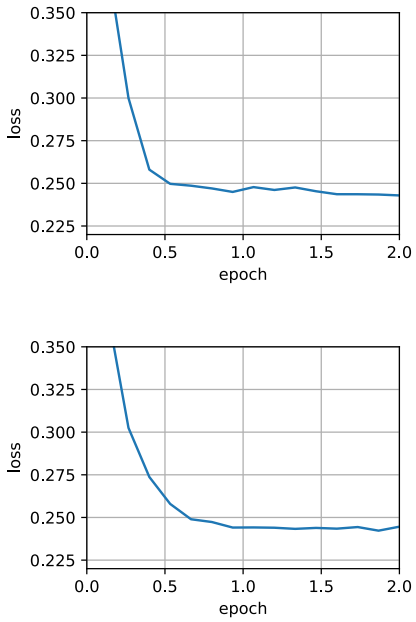

```
d2l.train_concise_ch11('adam', {'learning_rate': 0.01}, data_iter)
```

```
loss: 0.243, 0.352 sec/epoch
```

```
trainer = tf.keras.optimizers.Adam
d2l.train_concise_ch11(trainer, {'learning_rate': 0.01}, data_iter)
```

```
loss: 0.245, 1.688 sec/epoch
```

15.10.3 Yogi



Adam の問題の 1 つは、$\mathbf{s}_t$ における 2 次モーメント推定が大きくなりすぎると、凸最適化の設定でさえ収束に失敗しうることである。これに対する修正として、Zaheer *et al.* (2018) は $\mathbf{s}_t$ の更新（および初期化）を改良したものを提案した。何が起きているのかを理解するために、Adam の更新を次のように書き直してみよう。

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1-\beta_2)\left(\mathbf{g}_t^2 - \mathbf{s}_{t-1}\right). \tag{15.10.5}$$

$\mathbf{g}_t^2$ の分散が大きい場合や更新が疎な場合、$\mathbf{s}_t$ は過去の値をあまりにも速く忘れてしまうことがある。これに対する一つの修正は、$\mathbf{g}_t^2 - \mathbf{s}_{t-1}$ を $\mathbf{g}_t^2 \odot \mathrm{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1})$ に置き換えることである。すると、更新の大きさはもはや偏差の大きさに依存しない。これにより Yogi の更新式が得られる。

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \odot \mathrm{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1}). \tag{15.10.6}$$

著者らはさらに、モーメンタムを単なる初期の点ごとの推定ではなく、より大きな初期バッチで初期化することを勧めている。ここでは、議論の本質ではなく、またそれがなくても収束は十分良好なので、詳細は省略する。

```
def yogi(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-3
    for p, (v, s) in zip(params, states):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
            s[:] = s + (1 - beta2) * torch.sign(
                torch.square(p.grad) - s) * torch.square(p.grad)
            v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
```

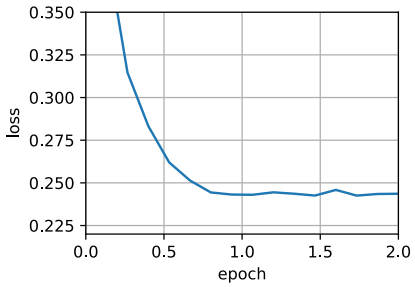
(continues on next page)

(continued from previous page)

```
            s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
            p[:] -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr)
                                                       + eps)
        p.grad.data.zero_()
    hyperparams['t'] += 1

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(yogi, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

```
loss: 0.244, 0.148 sec/epoch
```



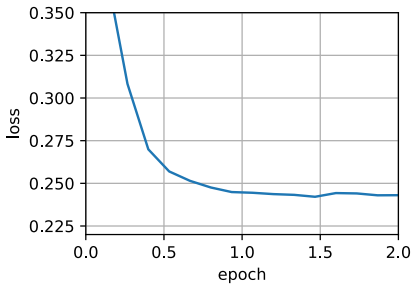

```
def yogi(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-3
    for p, (v, s) in zip(params, states):
        v[:] = beta1 * v + (1 - beta1) * p.grad
        s[:] = s + (1 - beta2) * np.sign(
            np.square(p.grad) - s) * np.square(p.grad)
        v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
        s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
        p[:] -= hyperparams['lr'] * v_bias_corr / (np.sqrt(s_bias_corr) + eps)
    hyperparams['t'] += 1

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(yogi, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

```
loss: 0.243, 0.795 sec/epoch
```

```
def yogi(params, grads, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s), grad in zip(params, states, grads):
        v[:].assign(beta1 * v  + (1 - beta1) * grad)
        s[:].assign(s + (1 - beta2) * tf.math.sign(
                    tf.math.square(grad) - s) * tf.math.square(grad))
        v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
        s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
```

(continues on next page)

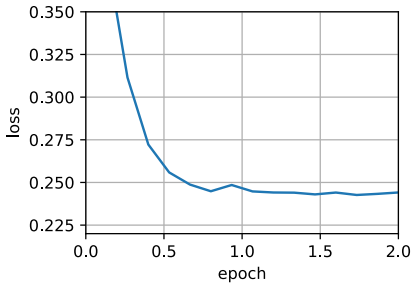


(continued from previous page)

```
        p[:].assign(p - hyperparams['lr'] * v_bias_corr
                    / tf.math.sqrt(s_bias_corr) + eps)
    hyperparams['t'] += 1

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(yogi, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

```
loss: 0.244, 1.759 sec/epoch
```



15.10.4 まとめ

- Adam は多くの最適化アルゴリズムの特徴を組み合わせた、かなり頑健な更新規則である。
- RMSProp を基に作られており、Adam はミニバッチ確率勾配にも EWMA を用いる。
- Adam は、モーメンタムと 2 次モーメントを推定する際の立ち上がりの遅さを調整するためにバイアス補正を使う。
- 分散の大きい勾配では、収束に問題が生じることがある。これは、より大きなミニバッチを使うか、$\mathbf{s}_t$ の改良された推定に切り替えることで改善できる。Yogi はそのような代替手法を提供する。

15.10.5 演習

1. 学習率を調整し、実験結果を観察して分析せよ。
2. バイアス補正を必要としないように、モーメンタムと 2 次モーメントの更新を書き換えられるか。
3. 収束に伴って学習率 η を下げる必要があるのはなぜか。
4. Adam が発散し、Yogi が収束する例を構成してみよ。

15.11 学習率スケジューリング

これまで主に、重みベクトルをどのように更新するかという最適化のアルゴリズムに注目してきたが、更新される速度についてはあまり扱ってこなかった。とはいえ、学習率の調整は実際のアルゴリズムと同じくらい重要であることがよくある。考慮すべき点はいくつかある。

- まず明らかなのは、学習率の大きさが重要だということである。大きすぎれば最適化は発散し、小さすぎれば学習に時間がかかりすぎるか、あるいは準最適な結果に終わってしまう。以前、問題の条件数が重要であることを見た（詳細は例えば Section 15.6 を参照）。直感的には、最も感度の低い方向における変化量と、最も感度の高い方向における変化量の比である。
- 第二に、減衰の速さも同様に重要である。学習率が大きいままだと、単に最小値の周りを行ったり来たりするだけで、最適性に到達できないかもしれない。Section 15.5 でこれを詳しく議論し、Section 15.4 では性能保証を解析した。要するに、学習率は減衰してほしいのだが、凸問題に対しては良い選択となる $O(t^{-\frac{1}{2}})$ よりはおそらくゆっくり減衰するほうがよいだろう。
- もう一つ同じくらい重要なのが初期化である。これは、パラメータを最初にどう設定するか（詳細は Section 8.4 を参照）だけでなく、初期段階でそれらがどう変化するかにも関係する。これはウォームアップという名目で扱われ、つまり最初に解へ向かってどれだけ速く動き始めるかということである。特に、初期のパラメータ設定はランダムなので、最初の大きなステップは有益でないかもしれない。初期の更新方向も、かなり無意味である可能性がある。
- 最後に、周期的に学習率を調整する最適化の変種もいくつかある。これは本章の範囲を超える。読者には Izmailov *et al.* (2018) の詳細、たとえばパラメータの全経路にわたって平均を取ることでより良い解を得る方法などを参照することを勧める。

学習率の管理には多くの詳細が必要であるため、ほとんどの深層学習フレームワークにはこれを自動的に扱うためのツールがある。本章では、異なるスケジ

ュールが精度に与える影響を確認し、さらにこれを学習率スケジューラによって効率的に管理する方法を示す。

15.11.1 おもちゃ問題

まず、計算コストが低く簡単に扱える一方で、いくつかの重要な点を示すのに十分に非自明なおもちゃ問題から始める。そのために、Fashion-MNIST に適用した LeNet の少し現代的な版（活性化関数は `sigmoid` ではなく `relu`、AveragePooling ではなく MaxPooling）を用いる。さらに、性能向上のためにネットワークをハイブリダイズする。コードの大部分は標準的なので、ここでは基本だけを紹介し、詳細な議論はしない。必要に応じて Section 10 を復習されたい。

```
%matplotlib inline
from d2l import torch as d2l
import math
import torch
from torch import nn
from torch.optim import lr_scheduler

def net_fn():
    model = nn.Sequential(
        nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(6, 16, kernel_size=5), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.Linear(16 * 5 * 5, 120), nn.ReLU(),
        nn.Linear(120, 84), nn.ReLU(),
        nn.Linear(84, 10))

    return model

loss = nn.CrossEntropyLoss()
device = d2l.try_gpu()

batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

# The code is almost identical to `d2l.train_ch6` defined in the
# lenet section of chapter convolutional neural networks
def train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
          scheduler=None):
    net.to(device)
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])

    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            net.train()
            trainer.zero_grad()
            X, y = X.to(device), y.to(device)
```

(continues on next page)

(continued from previous page)

```
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            trainer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            train_loss = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % 50 == 0:
                animator.add(epoch + i / len(train_iter),
                             (train_loss, train_acc, None))

        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch+1, (None, None, test_acc))

        if scheduler:
            if scheduler.__module__ == lr_scheduler.__name__:
                # Using PyTorch In-Built scheduler
                scheduler.step()
            else:
                # Using custom defined scheduler
                for param_group in trainer.param_groups:
                    param_group['lr'] = scheduler(epoch)

    print(f'train loss {train_loss:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, lr_scheduler, np, npx
from mxnet.gluon import nn
npx.set_np()

net = nn.HybridSequential()
net.add(nn.Conv2D(channels=6, kernel_size=5, padding=2, activation='relu'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(channels=16, kernel_size=5, activation='relu'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120, activation='relu'),
        nn.Dense(84, activation='relu'),
        nn.Dense(10))
net.hybridize()
loss = gluon.loss.SoftmaxCrossEntropyLoss()
device = d2l.try_gpu()

batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

# The code is almost identical to `d2l.train_ch6` defined in the
# lenet section of chapter convolutional neural networks
def train(net, train_iter, test_iter, num_epochs, loss, trainer, device):
    net.initialize(force_reinit=True, ctx=device, init=init.Xavier())
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
```

(continues on next page)

(continued from previous page)

```
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            X, y = X.as_in_ctx(device), y.as_in_ctx(device)
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y)
            l.backward()
            trainer.step(X.shape[0])
            metric.add(l.sum(), d2l.accuracy(y_hat, y), X.shape[0])
            train_loss = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % 50 == 0:
                animator.add(epoch + i / len(train_iter),
                             (train_loss, train_acc, None))
        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'train loss {train_loss:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
```

```
[07:52:47] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
class SquareRootScheduler:
    def __init__(self, lr=0.1):
        self.lr = lr

    def __call__(self, num_update):
        return self.lr * pow(num_update + 1.0, -0.5)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf
import math
from tensorflow.keras.callbacks import LearningRateScheduler

def net():
    return tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(filters=6, kernel_size=5, activation='relu',
                               padding='same'),
        tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
        tf.keras.layers.Conv2D(filters=16, kernel_size=5,
                               activation='relu'),
        tf.keras.layers.AvgPool2D(pool_size=2, strides=2),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(120, activation='relu'),
        tf.keras.layers.Dense(84, activation='sigmoid'),
        tf.keras.layers.Dense(10)])


batch_size = 256
```

(continues on next page)

(continued from previous page)

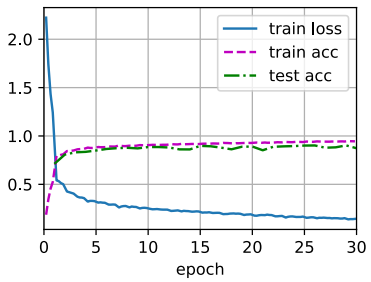
```
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

# The code is almost identical to `d2l.train_ch6` defined in the
# lenet section of chapter convolutional neural networks
def train(net_fn, train_iter, test_iter, num_epochs, lr,
              device=d2l.try_gpu(), custom_callback = False):
    device_name = device._device_name
    strategy = tf.distribute.OneDeviceStrategy(device_name)
    with strategy.scope():
        optimizer = tf.keras.optimizers.SGD(learning_rate=lr)
        loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
        net = net_fn()
        net.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])
    callback = d2l.TrainCallback(net, train_iter, test_iter, num_epochs,
                                 device_name)
    if custom_callback is False:
        net.fit(train_iter, epochs=num_epochs, verbose=0,
                callbacks=[callback])
    else:
         net.fit(train_iter, epochs=num_epochs, verbose=0,
                 callbacks=[callback, custom_callback])
    return net
```

このアルゴリズムを、学習率 0.3 で 30 イテレーション学習するというデフォルト設定で実行するとどうなるか見てみよう。訓練精度は上がり続ける一方で、テスト精度の向上はある点を超えると止まっていることに注意されたい。両方の曲線の間のギャップは過学習を示している。

```
lr, num_epochs = 0.3, 30
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=lr)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device)
```

```
train loss 0.145, train acc 0.943, test acc 0.873
```



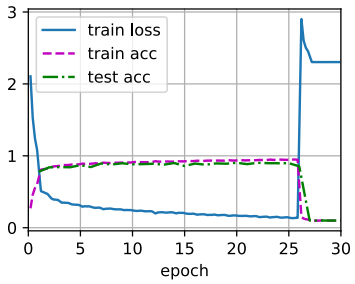

```
lr, num_epochs = 0.3, 30
net.initialize(force_reinit=True, ctx=device, init=init.Xavier())
```

(continues on next page)

(continued from previous page)

```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
train(net, train_iter, test_iter, num_epochs, loss, trainer, device)
```

```
train loss 2.303, train acc 0.099, test acc 0.100
```

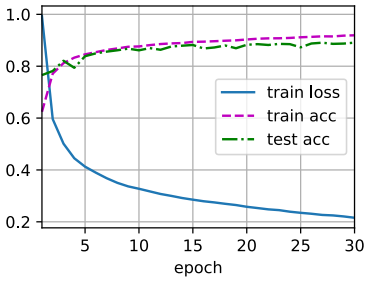



```
scheduler = SquareRootScheduler(lr=0.1)
d2l.plot(d2l.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```

```
lr, num_epochs = 0.3, 30
train(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.216, train acc 0.919, test acc 0.891
51230.9 examples/sec on /GPU:0
```

```
<keras.engine.sequential.Sequential at 0x7ff7b0148130>
```



15.11.2 スケジューラ

学習率を調整する一つの方法は、各ステップで明示的に設定することである。これは `set_learning_rate` メソッドで簡単に実現できる。たとえば、最適化の進み具合に応じて動的に、各エポック後（あるいは各ミニバッチ後）に学習率を下げることができる。

```
lr = 0.1
trainer.param_groups[0]["lr"] = lr
print(f'learning rate is now {trainer.param_groups[0]["lr"]:.2f}')
```

```
learning rate is now 0.10
```

```
trainer.set_learning_rate(0.1)
print(f'learning rate is now {trainer.learning_rate:.2f}')
```

```
learning rate is now 0.10
```

```
class FactorScheduler:
    def __init__(self, factor=1, stop_factor_lr=1e-7, base_lr=0.1):
        self.factor = factor
        self.stop_factor_lr = stop_factor_lr
        self.base_lr = base_lr

    def __call__(self, num_update):
        self.base_lr = max(self.stop_factor_lr, self.base_lr * self.factor)
        return self.base_lr

scheduler = FactorScheduler(factor=0.9, stop_factor_lr=1e-2, base_lr=2.0)
d2l.plot(d2l.arange(50), [scheduler(t) for t in range(50)])
```

```
lr = 0.1
dummy_model = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])
dummy_model.compile(tf.keras.optimizers.SGD(learning_rate=lr), loss='mse')
print(f'learning rate is now ,', dummy_model.optimizer.lr.numpy())
```

```
learning rate is now , 0.1
```

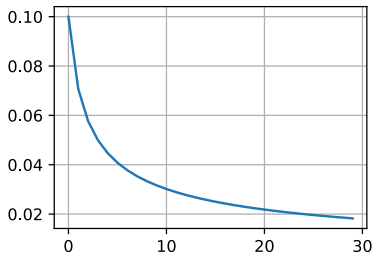
より一般的には、スケジューラを定義したいところである。更新回数を与えて呼び出すと、適切な学習率の値を返す。単純な例として、学習率を $\eta = \eta_0(t+1)^{-\frac{1}{2}}$ に設定するものを定義しよう。

```
class SquareRootScheduler:
    def __init__(self, lr=0.1):
        self.lr = lr

    def __call__(self, num_update):
        return self.lr * pow(num_update + 1.0, -0.5)
```

その振る舞いをいくつかの値にわたって描画してみよう。

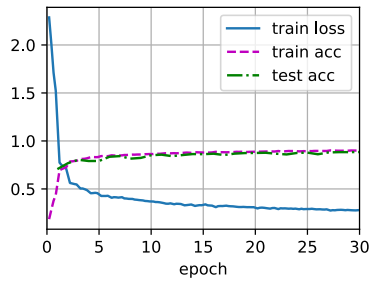
```
scheduler = SquareRootScheduler(lr=0.1)
d2l.plot(d2l.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



では、Fashion-MNIST で学習したときにこれがどう働くかを見てみよう。学習アルゴリズムに追加引数としてスケジューラを与えるだけである。

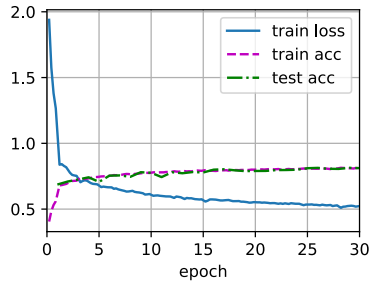
```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.279, train acc 0.899, test acc 0.884
```




```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, device)
```

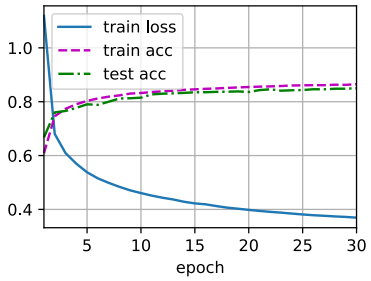
```
train loss 0.522, train acc 0.812, test acc 0.811
```



```
train(net, train_iter, test_iter, num_epochs, lr,
      custom_callback=LearningRateScheduler(scheduler))
```

```
loss 0.369, train acc 0.865, test acc 0.850
49886.9 examples/sec on /GPU:0
```

```
<keras.engine.sequential.Sequential at 0x7ff750362c20>
```



これは以前よりかなりうまくいった。注目すべき点が二つある。まず、曲線が以前よりかなり滑らかになった。第二に、過学習が少なくなった。残念ながら、なぜ特定の戦略が理論上過学習を減らすのかは、まだ十分に解明されていない。ステップサイズが小さいほど、ゼロに近いパラメータに到達しやすくなり、その結果としてより単純になる、という議論はある。しかし、実際には早期停止しているわけではなく、単に学習率を穏やかに下げているだけなので、この現象を完全には説明できない。

15.11.3 ポリシー

学習率スケジューラのあらゆる種類を網羅することはできないが、以下ではよく使われるポリシーを簡単に概観する。一般的な選択肢としては、多項式減衰と区分定数スケジュールがある。それに加えて、コサイン学習率スケジュールは、いくつかの問題で経験的によく機能することが分かっている。最後に、問題によっては、大きな学習率を使う前に最適化器をウォームアップすることが有益である。

Factor Scheduler

多項式減衰の代わりに、乗法的な減衰、すなわち $\alpha \in (0, 1)$ に対して $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ を用いる方法がある。学習率が妥当な下限を超えて減衰しないようにするため、更新式はしばしば $\eta_{t+1} \leftarrow \max(\eta_{\min}, \eta_t \cdot \alpha)$ に修正される。

```
class FactorScheduler:
    def __init__(self, factor=1, stop_factor_lr=1e-7, base_lr=0.1):
```

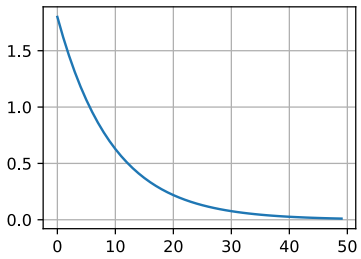
(continues on next page)

(continued from previous page)

```
        self.factor = factor
        self.stop_factor_lr = stop_factor_lr
        self.base_lr = base_lr

    def __call__(self, num_update):
        self.base_lr = max(self.stop_factor_lr, self.base_lr * self.factor)
        return self.base_lr

scheduler = FactorScheduler(factor=0.9, stop_factor_lr=1e-2, base_lr=2.0)
d2l.plot(d2l.arange(50), [scheduler(t) for t in range(50)])
```



これは MXNet では `lr_scheduler.FactorScheduler` オブジェクトという組み込みスケジューラでも実現できる。これには、ウォームアップ期間、ウォームアップのモード（線形または定数）、望ましい更新回数の上限など、いくつか追加のパラメータがある。今後は、適宜組み込みスケジューラを使い、その機能についてここで説明することにする。図に示したように、必要なら自分でスケジューラを作るのはかなり簡単である。

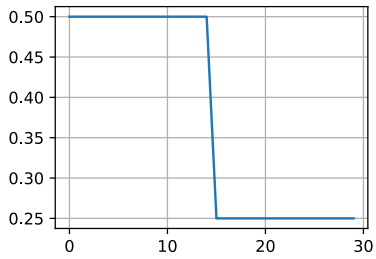
Multi Factor Scheduler

深層ネットワークの学習でよくある戦略は、学習率を区分的に一定に保ち、一定間隔で所定の量だけ下げることである。つまり、たとえば $s = \{5, 10, 20\}$ のように学習率を下げる時刻の集合が与えられたとき、$t \in s$ ならば $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ とする。各段階で半分にする場合は、次のように実装できる。

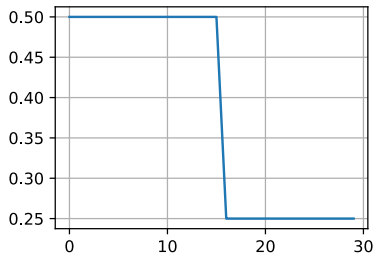
```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.5)
scheduler = lr_scheduler.MultiStepLR(trainer, milestones=[15, 30], gamma=0.5)

def get_lr(trainer, scheduler):
    lr = scheduler.get_last_lr()[0]
    trainer.step()
    scheduler.step()
    return lr

d2l.plot(d2l.arange(num_epochs), [get_lr(trainer, scheduler)
                                  for t in range(num_epochs)])
```



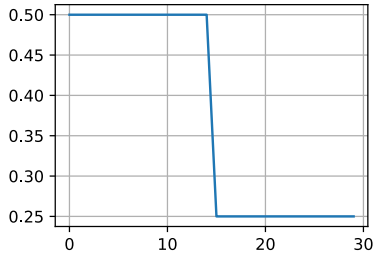

```
scheduler = lr_scheduler.MultiFactorScheduler(step=[15, 30], factor=0.5,
                                              base_lr=0.5)
d2l.plot(d2l.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```




```
class MultiFactorScheduler:
    def __init__(self, step, factor, base_lr):
        self.step = step
        self.factor = factor
        self.base_lr = base_lr

    def __call__(self, epoch):
        if epoch in self.step:
            self.base_lr = self.base_lr * self.factor
            return self.base_lr
        else:
            return self.base_lr

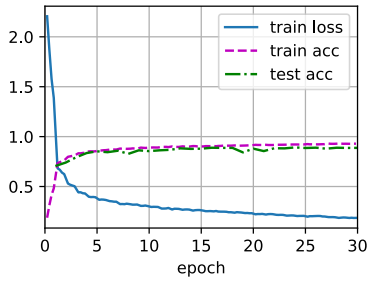
scheduler = MultiFactorScheduler(step=[15, 30], factor=0.5, base_lr=0.5)
d2l.plot(d2l.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



この区分定数の学習率スケジュールの背後にある直感は、重みベクトルの分布の観点で定常点に達するまで最適化を進め、その後でのみ学習率を下げることで、良い局所最小値のより高品質な近似を得るというものである。以下の例では、これがわずかにより良い解を生み出す様子を示している。

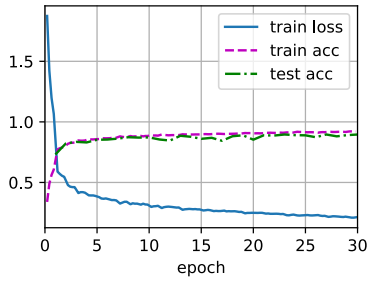
```
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.186, train acc 0.929, test acc 0.887
```




```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, device)
```

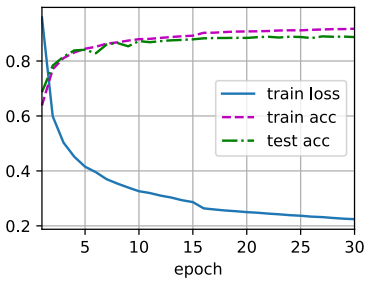
```
train loss 0.213, train acc 0.920, test acc 0.896
```




```
train(net, train_iter, test_iter, num_epochs, lr,
      custom_callback=LearningRateScheduler(scheduler))
```

```
loss 0.224, train acc 0.917, test acc 0.887
50810.6 examples/sec on /GPU:0
```

```
<keras.engine.sequential.Sequential at 0x7ff750335bd0>
```



Cosine Scheduler

Loshchilov and Hutter (2016) によって、やや意外なヒューリスティックが提案された。これは、最初に学習率をあまり急激に下げたくないこと、さらに最後には非常に小さな学習率を使って解を「洗練」したいこと、という観察に基づいている。その結果、$t \in [0, T]$ の範囲で学習率が次のようなコサイン状のスケジュールになる。

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2} (1 + \cos(\pi t / T)) \tag{15.11.1}$$

ここで η_0 は初期学習率、η_T は時刻 T における目標学習率である。さらに、$t > T$ では値を再び増やさず、単に η_T に固定する。以下の例では、最大更新ステップを $T = 20$ に設定する。

```
class CosineScheduler:
    def __init__(self, max_update, base_lr=0.01, final_lr=0,
                 warmup_steps=0, warmup_begin_lr=0):
        self.base_lr_orig = base_lr
        self.max_update = max_update
        self.final_lr = final_lr
        self.warmup_steps = warmup_steps
        self.warmup_begin_lr = warmup_begin_lr
        self.max_steps = self.max_update - self.warmup_steps

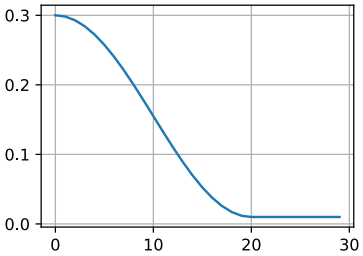
    def get_warmup_lr(self, epoch):
        increase = (self.base_lr_orig - self.warmup_begin_lr) \
                       * float(epoch) / float(self.warmup_steps)
        return self.warmup_begin_lr + increase

    def __call__(self, epoch):
        if epoch < self.warmup_steps:
            return self.get_warmup_lr(epoch)
        if epoch <= self.max_update:
            self.base_lr = self.final_lr + (
                self.base_lr_orig - self.final_lr) * (1 + math.cos(
                math.pi * (epoch - self.warmup_steps) / self.max_steps)) / 2
        return self.base_lr
```

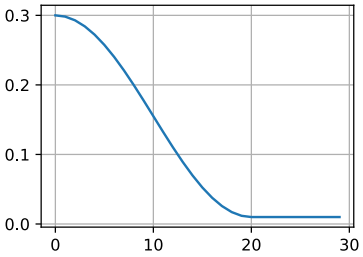
(continues on next page)

(continued from previous page)

```
scheduler = CosineScheduler(max_update=20, base_lr=0.3, final_lr=0.01)
d2l.plot(d2l.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```




```
scheduler = lr_scheduler.CosineScheduler(max_update=20, base_lr=0.3,
                                         final_lr=0.01)
d2l.plot(d2l.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```




```
class CosineScheduler:
    def __init__(self, max_update, base_lr=0.01, final_lr=0,
                 warmup_steps=0, warmup_begin_lr=0):
        self.base_lr_orig = base_lr
        self.max_update = max_update
        self.final_lr = final_lr
        self.warmup_steps = warmup_steps
        self.warmup_begin_lr = warmup_begin_lr
        self.max_steps = self.max_update - self.warmup_steps

    def get_warmup_lr(self, epoch):
        increase = (self.base_lr_orig - self.warmup_begin_lr) \
                       * float(epoch) / float(self.warmup_steps)
        return self.warmup_begin_lr + increase

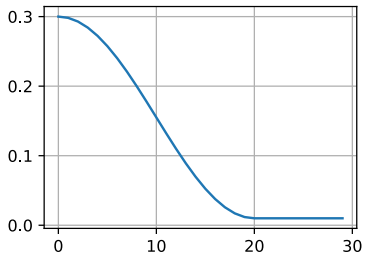
    def __call__(self, epoch):
        if epoch < self.warmup_steps:
            return self.get_warmup_lr(epoch)
        if epoch <= self.max_update:
            self.base_lr = self.final_lr + (
                self.base_lr_orig - self.final_lr) * (1 + math.cos(
```

(continues on next page)

(continued from previous page)

```
                math.pi * (epoch - self.warmup_steps) / self.max_steps)) / 2
        return self.base_lr

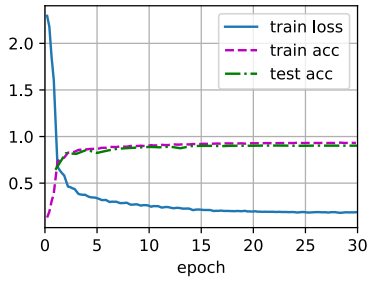
scheduler = CosineScheduler(max_update=20, base_lr=0.3, final_lr=0.01)
d2l.plot(d2l.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



コンピュータビジョンの文脈では、このスケジュールは結果を改善することがある。ただし、そのような改善が保証されるわけではない（下に示すとおりである）。

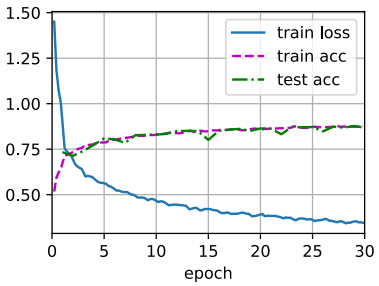
```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.187, train acc 0.931, test acc 0.902
```




```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, device)
```

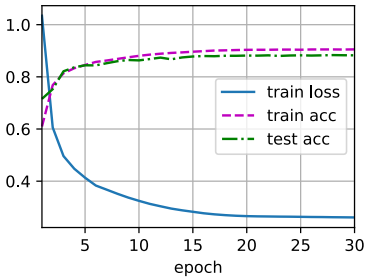
```
train loss 0.345, train acc 0.877, test acc 0.868
```




```
train(net, train_iter, test_iter, num_epochs, lr,
      custom_callback=LearningRateScheduler(scheduler))
```

```
loss 0.261, train acc 0.905, test acc 0.882
50105.5 examples/sec on /GPU:0
```

```
<keras.engine.sequential.Sequential at 0x7ff768221210>
```

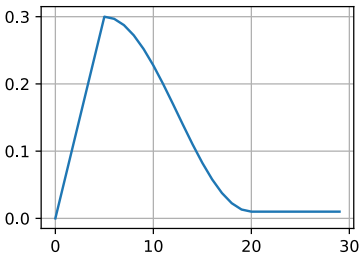


ウォームアップ

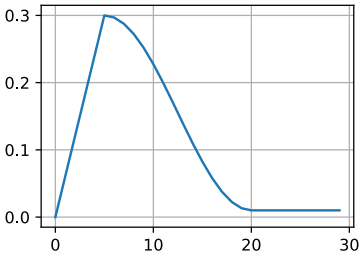
場合によっては、パラメータの初期化だけでは良い解を保証するのに十分ではない。これは特に、最適化が不安定になりうる高度なネットワーク設計で問題になる。これに対処するには、最初のうち発散を防ぐために十分小さい学習率を選ぶ方法がある。残念ながら、これは進みが遅いことを意味する。逆に、最初から大きな学習率を使うと発散する。

このジレンマに対するかなり単純な解決策は、学習率が初期の最大値まで増加するウォームアップ期間を設け、その後は最適化過程の終わりまで学習率を下げていくことである。簡単のため、通常はこの目的に線形増加を用いる。これにより、以下に示すようなスケジュールになる。

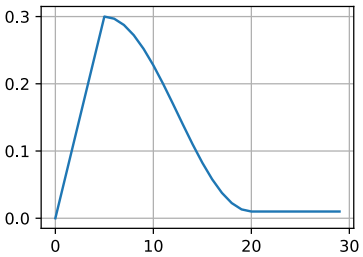
```
scheduler = CosineScheduler(20, warmup_steps=5, base_lr=0.3, final_lr=0.01)
d2l.plot(d2l.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```




```
scheduler = lr_scheduler.CosineScheduler(20, warmup_steps=5, base_lr=0.3,
                                         final_lr=0.01)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



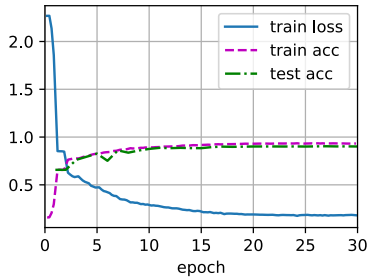

```
scheduler = CosineScheduler(20, warmup_steps=5, base_lr=0.3, final_lr=0.01)
d2l.plot(d2l.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



ネットワークが初期段階でよりよく収束することに注意されたい（特に最初の 5 エポックでの性能を観察されたい）。

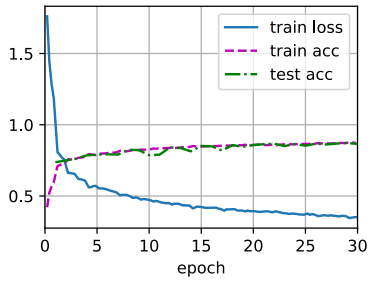
```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.181, train acc 0.933, test acc 0.901
```




```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, device)
```

```
train loss 0.352, train acc 0.873, test acc 0.865
```

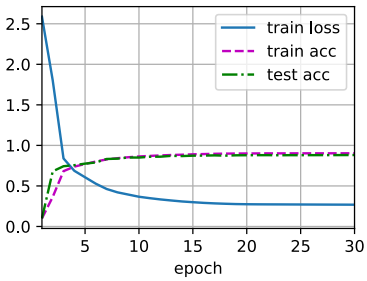



```
train(net, train_iter, test_iter, num_epochs, lr,
      custom_callback=LearningRateScheduler(scheduler))
```

```
loss 0.269, train acc 0.903, test acc 0.881
51441.5 examples/sec on /GPU:0
```

```
<keras.engine.sequential.Sequential at 0x7ff7c3509600>
```

ウォームアップは任意のスケジューラに適用できる（コサインに限らない）。学習率スケジュールについてのより詳細な議論や、さらに多くの実験については (Gotmare *et al.*, 2018) も参照されたい。特に、ウォームアップ段階は非常に深いネットワークにおけるパラメータの発散量を抑えることが分かっている。これは直感的にも理解できる。というのも、学習の初期に進展しにくいネットワーク部分では、ランダム初期化によって大きな発散が生じると予想されるからである。



15.11.4 まとめ

- 学習中に学習率を下げると、精度の向上や、（最も不可解なことに）モデルの過学習の減少につながることがある。
- 進展が頭打ちになったときに学習率を段階的に下げる方法は、実際に有効である。本質的には、これにより適切な解へ効率よく収束し、その後でのみ学習率を下げることでパラメータの本質的な分散を減らす。
- コサインスケジューラは、いくつかのコンピュータビジョン問題で人気がある。このようなスケジューラの詳細は、例えば GluonCV[90] を参照されたい。
- 最適化の前にウォームアップ期間を設けると、発散を防げる。
- 最適化は深層学習において複数の目的を果たす。訓練目的関数の最小化だけでなく、最適化アルゴリズムや学習率スケジューリングの選択によって、（同じ訓練誤差でも）テストセット上での汎化や過学習の程度がかなり異なることがある。

90

15.11.5 演習

1. 固定した学習率に対する最適化の挙動を実験せよ。この方法で得られる最良のモデルは何か。
2. 学習率の減衰指数を変えると収束はどう変わるか。実験では `PolyScheduler` を使うと便利である。
3. コサインスケジューラを大規模なコンピュータビジョン問題、たとえば ImageNet の学習に適用せよ。他のスケジューラと比べて性能にどのような影響があるか。
4. ウォームアップはどのくらいの長さにすべきか。
5. 最適化とサンプリングを結びつけられるか。まずは Welling and Teh (2011) の確率的勾配ランジュバン動力学に関する結果を使ってみよ。

16 計算性能

深層学習では、
データセットやモデルは通常大規模であり、
そのため重い計算を伴いる。
したがって、計算性能は非常に重要である。
この章では、計算性能に影響を与える主な要因、すなわち、命令型プログラミング、記号型プログラミング、非同期計算、自動並列化、およびマルチ GPU 計算に焦点を当てる。
この章を学ぶことで、前の章で実装したこれらのモデルの計算性能をさらに向上させることができる。
たとえば、精度を損なうことなく学習時間を短縮する、といった改善が可能である。

16.1 コンパイラとインタプリタ

これまで本書では、`print`、`+`、`if` などの文を用いてプログラムの状態を変更する命令型プログラミングに焦点を当ててきた。以下に、単純な命令型プログラムの例を示す。

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))
```

```
10
```

Python は インタプリタ型言語である。上の `fancy_func` 関数を評価するとき、関数本体を構成する操作を 順番に実行する。つまり、`e = add(a, b)` を評価して結

果を変数 e に保存し、それによってプログラムの状態を変更する。続く 2 つの文 f = add(c, d) と g = add(e, f) も同様に実行され、加算が行われて結果が変数として保存される。Fig. 16.1.1 はデータの流れを示している。

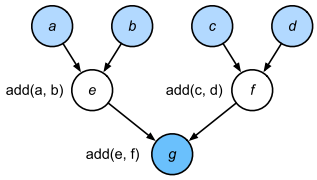


Fig. 16.1.1 命令型プログラムにおけるデータフロー。

命令型プログラミングは便利であるが、非効率な場合がある。ひとつには、fancy_func 全体で add 関数が繰り返し呼び出されても、Python は 3 回の関数呼び出しを個別に実行するからである。これらが、たとえば GPU 上（あるいは複数 GPU 上）で実行される場合、Python インタプリタに起因するオーバーヘッドが非常に大きくなることがある。さらに、fancy_func 内のすべての文が実行されるまで、変数 e と f の値を保存しておく必要がある。これは、文 e = add(a, b) と f = add(c, d) が実行された後に、変数 e と f がプログラムの他の部分で使われるかどうかが分からないためである。

16.1.1 シンボリックプログラミング

代替案として シンボリックプログラミングを考えよう。これは通常、処理が完全に定義されてから計算を行う方式である。この戦略は、Theano や TensorFlow（後者は命令型拡張を取り込んでいます）を含む複数の深層学習フレームワークで使われている。通常、次の手順を含む。

1. 実行する操作を定義する。
2. 操作を実行可能なプログラムにコンパイルする。
3. 必要な入力を与え、コンパイル済みプログラムを呼び出して実行する。

これにより、大幅な最適化が可能になる。まず、多くの場合 Python インタプリタを省略できるため、単一の Python スレッドが CPU 上で動作しながら複数の高速 GPU を使うような状況で顕著になりうる性能ボトルネックを取り除ける。
第二に、コンパイラは上のコードを print((1 + 2) + (3 + 4))、あるいは print(10) にまで最適化して書き換えられるかもしれない。これは、コンパイラが機械命令に変換する前にコード全体を見渡せるからである。たとえば、ある変数が不要になった時点でメモリを解放したり（あるいは最初から確保しなかったり）できる。また、コード全体を等価な別のコードへ変換することもできる。
よりよく理解するために、以下の命令型プログラミングのシミュレーション（結局のところ Python です）を見てみよう。

```
def add_():
    return '''
def add(a, b):
    return a + b
'''

def fancy_func_():
    return '''
def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
'''

def evoke_():
    return add_() + fancy_func_() + 'print(fancy_func(1, 2, 3, 4))'

prog = evoke_()
print(prog)
y = compile(prog, '', 'exec')
exec(y)
```

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
print(fancy_func(1, 2, 3, 4))
10
```

命令型（インタプリタ型）プログラミングとシンボリックプログラミングの違いは次のとおりである。

- 命令型プログラミングは簡単である。Python で命令型プログラミングを使う場合、コードの大部分は素直で書きやすいものである。また、命令型プログラミングのコードはデバッグしやすいである。これは、関連する中間変数の値をすべて取得して表示したり、Python に組み込まれたデバッグツールを使ったりしやすいためである。

- シンボリックプログラミングはより効率的で、移植もしやすいである。シンボリックプログラミングでは、コンパイル中にコードを最適化しやすいだけでなく、Python に依存しない形式へプログラムを移植することもできる。これにより、Python 以外の環境でもプログラムを実行でき、Python インタプリタに関連する潜在的な性能問題を回避できる。

16.1.2 ハイブリッドプログラミング

歴史的に、多くの深層学習フレームワークは命令型かシンボリック型のどちらかを選んできた。たとえば、Theano、TensorFlow（前者に触発された）、Keras、CNTK はモデルをシンボリックに記述する。逆に、Chainer と PyTorch は命令型アプローチを採用している。TensorFlow 2.0 と Keras には、後の改訂で命令型モードが追加された。

上で述べたように、PyTorch は命令型プログラミングに基づいており、動的計算グラフを使用する。シンボリックプログラミングの移植性と効率性を活用するため、開発者たちは両方のプログラミングパラダイムの利点を組み合わせられるかどうかを検討した。その結果、ユーザーは純粋な命令型プログラミングで開発とデバッグを行いながら、製品レベルの計算性能やデプロイが必要なときには、ほとんどのプログラムをシンボリックプログラムへ変換して実行できる torchscript が生まれた。

16.1.3 `Sequential` クラスのハイブリッド化

ハイブリッド化の仕組みを理解する最も簡単な方法は、複数層を持つ深いネットワークを考えることである。従来は、Python インタプリタがすべての層のコードを実行して、CPU や GPU に転送できる命令を生成する必要があった。単一の（高速な）計算デバイスであれば、これは大きな問題にはならない。一方、AWS P3dn.24xlarge インスタンスのような高度な 8 GPU サーバーを使う場合、Python はすべての GPU を忙しく保つのに苦労する。ここでは単一スレッドの Python インタプリタがボトルネックになる。`Sequential` を `HybridSequential` に置き換えることで、コードのかなりの部分についてこの問題にどう対処できるか見てみよう。まず、単純な MLP を定義する。

```
from d2l import torch as d2l
import torch
from torch import nn

# Factory for networks
def get_net():
    net = nn.Sequential(nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 2))
    return net

x = torch.randn(size=(1, 512))
net = get_net()
net(x)
```

```
tensor([[0.1385, 0.1115]], grad_fn=<AddmmBackward0>)
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
```

(continues on next page)

(continued from previous page)

```
from mxnet.gluon import nn
npx.set_np()

# Factory for networks
def get_net():
    net = nn.HybridSequential()
    net.add(nn.Dense(256, activation='relu'),
            nn.Dense(128, activation='relu'),
            nn.Dense(2))
    net.initialize()
    return net

x = np.random.normal(size=(1, 512))
net = get_net()
net(x)
```

```
[07:00:25] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
array([[ 0.16526175, -0.14005634]])
```

```
#@save
class Benchmark:
    """For measuring running time."""
    def __init__(self, description='Done'):
        self.description = description

    def __enter__(self):
        self.timer = d2l.Timer()
        return self

    def __exit__(self, *args):
        print(f'{self.description}: {self.timer.stop():.4f} sec')
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
from tensorflow.keras.layers import Dense

# Factory for networks
def get_net():
    net = tf.keras.Sequential()
    net.add(Dense(256, input_shape = (512,), activation = "relu"))
    net.add(Dense(128, activation = "relu"))
    net.add(Dense(2, activation = "linear"))
    return net

x = tf.random.normal([1,512])
net = get_net()
net(x)
```

```
<tf.Tensor: shape=(1, 2), dtype=float32, numpy=array([[ 1.5561402, -1.552362␣
↪]], dtype=float32)>
```

torch.jit.script 関数を使ってモデルを変換することで、MLP 内の計算をコンパイルして最適化できる。モデルの計算結果は変わらない。

```
net = torch.jit.script(net)
net(x)
```

```
tensor([[0.1385, 0.1115]], grad_fn=<AddmmBackward0>)
```

```
net.hybridize()
net(x)
```

```
array([[ 0.16526175, -0.14005634]])
```

```
net = tf.function(net)
net(x)
```

```
<tf.Tensor: shape=(1, 2), dtype=float32, numpy=array([[ 1.5561402, -1.552362␣
↪]], dtype=float32)>
```

これは、あまりにも都合がよすぎるように見えるかもしれない。以前と同じコードを書き、単に torch.jit.script を使ってモデルを変換するだけである。こうするとネットワークは最適化される（性能は後でベンチマークする）。

ハイブリッド化による高速化

コンパイルによって得られる性能向上を示すために、ハイブリッド化の前後で net(x) の評価に必要な時間を比較する。まず、この時間を測定するクラスを定義しよう。これは、性能を測定し（そして改善し）ようとする本章全体で役立ちる。

```
#@save
class Benchmark:
    """For measuring running time."""
    def __init__(self, description='Done'):
        self.description = description

    def __enter__(self):
        self.timer = d2l.Timer()
        return self
```

(continues on next page)

(continued from previous page)

```
    def __exit__(self, *args):
        print(f'{self.description}: {self.timer.stop():.4f} sec')
```

では、ネットワークを 2 回呼び出してみよう。1 回は torchscript なし、もう 1 回は torchscript ありである。

```
net = get_net()
with Benchmark('Without torchscript'):
    for i in range(1000): net(x)

net = torch.jit.script(net)
with Benchmark('With torchscript'):
    for i in range(1000): net(x)
```

```
Without torchscript: 1.7613 sec
With torchscript: 3.0537 sec
```

```
net = get_net()
with Benchmark('Without hybridization'):
    for i in range(1000): net(x)
    npx.waitall()

net.hybridize()
with Benchmark('With hybridization'):
    for i in range(1000): net(x)
    npx.waitall()
```

```
Without hybridization: 0.6049 sec
With hybridization: 0.5470 sec
```

```
net = get_net()
with Benchmark('Eager Mode'):
    for i in range(1000): net(x)

net = tf.function(net)
with Benchmark('Graph Mode'):
    for i in range(1000): net(x)
```

```
Eager Mode: 1.9632 sec
Graph Mode: 0.4807 sec
```

上の結果から分かるように、`nn.Sequential` のインスタンスを `torch.jit.script` 関数でスクリプト化した後は、シンボリックプログラミングの利用によって計算性能が向上する。

シリアライズ

モデルをコンパイルする利点のひとつは、モデルとそのパラメータをディスクにシリアライズ（保存）できることである。これにより、選択したフロントエンド言語に依存しない形でモデルを保存できる。したがって、学習済みモデルを他のデバイスへデプロイしたり、他のフロントエンドプログラミング言語を簡単に使ったりできる。同時に、コードは命令型プログラミングで達成できるものよりもしばしば高速である。save 関数を見てみよう。

```
net.save('my_mlp')
!ls -lh my_mlp*
```

```
-rw-r--r-- 1 ci ci 651K Aug 18 07:11 my_mlp
```

```
net.export('my_mlp')
!ls -lh my_mlp*
```

```
-rw-r--r-- 1 ci ci 643K Aug 18 07:00 my_mlp-0000.params
-rw-r--r-- 1 ci ci 3.2K Aug 18 07:00 my_mlp-symbol.json
```

```
net = get_net()
tf.saved_model.save(net, 'my_mlp')
!ls -lh my_mlp*
```

```
INFO:tensorflow:Assets written to: my_mlp/assets
total 72K
drwxr-xr-x 2 ci ci   6 Aug 18 07:37 assets
-rw-r--r-- 1 ci ci  55 Aug 18 07:37 fingerprint.pb
-rw-r--r-- 1 ci ci 68K Aug 18 07:37 saved_model.pb
drwxr-xr-x 2 ci ci  66 Aug 18 07:37 variables
```

16.1.4 まとめ

- 命令型プログラミングでは、制御フローを使え、Python の豊富なソフトウェアエコシステムを活用できるため、新しいモデルを設計しやすいである。
- シンボリックプログラミングでは、実行前にプログラムを指定してコンパイルする必要がある。その利点は性能向上である。

16.1.5 演習

1. 以前の章で興味を持ったモデルを見直しよ。再実装することで計算性能を改善できるか。

16.2 非同期計算

今日のコンピュータは高度に並列なシステムであり、複数の CPU コア（多くの場合、各コアに複数のスレッド）、GPU あたり複数の処理要素、そしてしばしばデバイスあたり複数の GPU から構成されている。要するに、私たちは多くの異なる処理を同時に、しばしば異なるデバイス上で実行できる。残念ながら、Python は並列・非同期コードを書くのにあまり向いていない。少なくとも、何らかの追加の助けなしにはそうである。結局のところ、Python は単一スレッドであり、将来これが変わる可能性は低いだろう。MXNet や TensorFlow のような深層学習フレームワークは性能を向上させるために 非同期プログラミングモデルを採用しているが、PyTorch は Python 自身のスケジューラを利用しており、そのため性能上のトレードオフが異なる。PyTorch では、デフォルトで GPU 操作は非同期である。GPU を使う関数を呼び出すと、その操作は特定のデバイスにキューイングされるが、必ずしもすぐに実行されるとは限らない。これにより、CPU や他の GPU 上の操作を含め、より多くの計算を並列に実行できる。

したがって、非同期プログラミングの仕組みを理解することは、計算要件や相互依存を事前に減らすことで、より効率的なプログラムを開発する助けになる。これにより、メモリオーバーヘッドを削減し、プロセッサの利用率を高めることができる。

```
from d2l import torch as d2l
import numpy, os, subprocess
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
import numpy, os, subprocess
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
npx.set_np()
```

16.2.1 バックエンドによる非同期性

準備運動として、次の簡単な問題を考えよう。乱数行列を生成して、それを掛け合わせたいとする。NumPy と PyTorch のテンソルの両方でそれを行い、違いを見てみよう。なお、PyTorch の `tensor` は GPU 上に定義されている。

```
# Warmup for GPU computation
device = d2l.try_gpu()
a = torch.randn(size=(1000, 1000), device=device)
b = torch.mm(a, a)
```

(continues on next page)

(continued from previous page)

```
with d2l.Benchmark('numpy'):
    for _ in range(10):
        a = numpy.random.normal(size=(1000, 1000))
        b = numpy.dot(a, a)

with d2l.Benchmark('torch'):
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
```

```
numpy: 0.9937 sec
torch: 0.0019 sec
```

```
with d2l.Benchmark('numpy'):
    for _ in range(10):
        a = numpy.random.normal(size=(1000, 1000))
        b = numpy.dot(a, a)

with d2l.Benchmark('mxnet.np'):
    for _ in range(10):
        a = np.random.normal(size=(1000, 1000))
        b = np.dot(a, a)
```

```
numpy: 0.8821 sec
mxnet.np: 0.0222 sec
[07:01:48] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

PyTorch 経由のベンチマーク結果は桁違いに高速である。NumPy のドット積は CPU プロセッサ上で実行される一方、PyTorch の行列積は GPU 上で実行されるため、後者がはるかに高速であることは予想される。しかし、この大きな時間差は、何か別のことが起きていることを示唆している。デフォルトでは、PyTorch では GPU 操作は非同期である。PyTorch にすべての計算を返却前に完了させるよう強制すると、以前何が起きていたかが分かりる。つまり、フロントエンドが Python に制御を返している間に、計算はバックエンドで実行されているのである。

```
with d2l.Benchmark():
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
    torch.cuda.synchronize(device)
```

```
Done: 0.0025 sec
```

```
with d2l.Benchmark():
    for _ in range(10):
        a = np.random.normal(size=(1000, 1000))
        b = np.dot(a, a)
    npx.waitall()
```

```
Done: 1.1622 sec
```

大まかに言えば、PyTorch にはユーザーとの直接的なやり取りを行うフロントエンド（たとえば Python 経由）と、計算を実行するためにシステムが用いるバックエンドがある。Fig. 16.2.1 に示すように、ユーザーは Python や C++ など、さまざまなフロントエンド言語で PyTorch プログラムを書くことができる。どのフロントエンド言語を使っても、PyTorch プログラムの実行は主として C++ 実装のバックエンドで行われる。フロントエンド言語から発行された操作は、実行のためにバックエンドへ渡される。バックエンドは独自のスレッドを管理し、キューに入ったタスクを継続的に収集して実行する。これが機能するためには、バックエンドが計算グラフ内のさまざまなステップ間の依存関係を追跡できなければならないことに注意しよう。したがって、互いに依存する操作を並列化することはできない。

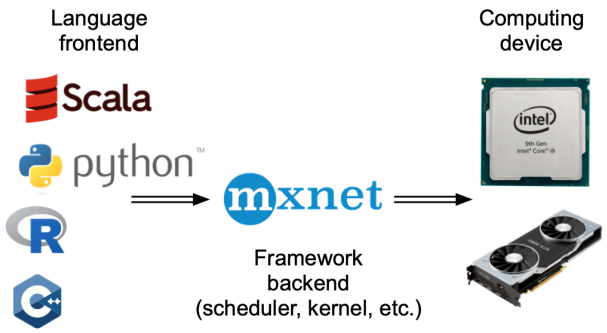


Fig. 16.2.1 Programming language frontends and deep learning framework backends.

依存グラフをもう少しよく理解するために、別の簡単な例を見てみよう。

```
x = torch.ones((1, 2), device=device)
y = torch.ones((1, 2), device=device)
z = x * y + 2
z
```

```
tensor([[3., 3.]], device='cuda:0')
```

```
x = np.ones((1, 2))
y = np.ones((1, 2))
```

(continues on next page)

(continued from previous page)

```
z = x * y + 2
z
```

```
array([[3., 3.]])
```

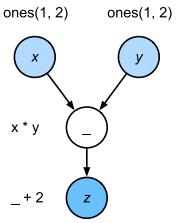


Fig. 16.2.2 The backend tracks dependencies between various steps in the computational graph.

上のコード片は Fig. 16.2.2 にも示されている。Python フロントエンドのスレッドが最初の 3 つの文のいずれかを実行すると、単にタスクをバックエンドのキューに返すだけである。最後の文の結果を 表示する必要があるとき、Python フロントエンドのスレッドは C++ バックエンドのスレッドが変数 z の結果の計算を終えるまで待機する。この設計の利点の 1 つは、Python フロントエンドのスレッドが実際の計算を行う必要がないことである。したがって、Python の性能にかかわらず、プログラム全体の性能への影響は小さくなる。Fig. 16.2.3 はフロントエンドとバックエンドの相互作用を示している。

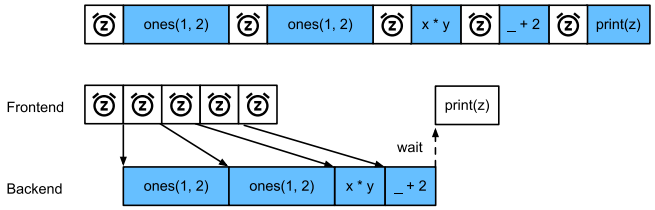


Fig. 16.2.3 Interactions of the frontend and backend.

16.2.2 バリアとブロッカー

16.2.3 計算の改善

16.2.4 まとめ

- 深層学習フレームワークは、Python フロントエンドを実行バックエンドから分離することがある。これにより、コマンドをバックエンドへ高速に非同期投入でき、それに伴う並列性が得られる。
- 非同期性により、フロントエンドはかなり応答性が高くなる。ただし、タスクキューを埋めすぎると過剰なメモリ消費につながる可能性があるため注

意が必要である。フロントエンドとバックエンドをおおむね同期させるために、ミニバッチごとに同期することが推奨される。

- チップベンダーは、深層学習の効率について、よりきめ細かな洞察を得るための高度な性能解析ツールを提供している。

16.2.5 演習

1. CPU 上で、この節と同じ行列積演算をベンチマークしなさい。バックエンドによる非同期性はまだ観測できるか。

16.3 自動並列化

ディープラーニング・フレームワーク（たとえば MXNet や PyTorch）は、バックエンドで計算グラフを自動的に構築する。計算グラフを用いることで、システムはすべての依存関係を把握でき、相互に依存しない複数のタスクを選択的に並列実行して速度を向上させることができる。たとえば、Section 16.2 の Fig. 16.2.2 では 2 つの変数を独立に初期化している。そのため、システムはそれらを並列に実行することを選べる。

通常、単一の演算子は、すべての CPU 上の計算資源、または単一の GPU 上の計算資源をすべて使う。たとえば dot 演算子は、1 台のマシンに複数の CPU プロセッサがあっても、すべての CPU の全コア（およびスレッド）を使う。GPU でも同様である。したがって、単一デバイスのコンピュータでは並列化の有用性はそれほど高くない。複数デバイスがある場合には、事情が変わりる。並列化は通常、複数 GPU 間で最も重要であるが、ローカル CPU を加えることで性能はわずかに向上する。たとえば、GPU と CPU を組み合わせてコンピュータビジョンモデルを学習することに焦点を当てた Hadjis *et al.* (2016) を参照しよ。自動並列化するフレームワークの利便性があれば、同じ目的を数行の Python コードで達成できる。より広く言えば、自動並列計算に関する本節の議論は、CPU と GPU の両方を用いた並列計算、および計算と通信の並列化に焦点を当てている。

なお、本節の実験を実行するには少なくとも 2 枚の GPU が必要である。

```
from d2l import torch as d2l
import torch
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()
```

16.3.1 GPU 上での並列計算

まず、テスト用の基準となるワークロードを定義しよう。以下の run 関数は、2 つの変数 x_gpu1 と x_gpu2 に割り当てられたデータを使って、選択したデバイス上で 10 回の行列積を実行する。

```
devices = d2l.try_all_gpus()
def run(x):
    return [x.mm(x) for _ in range(50)]

x_gpu1 = torch.rand(size=(4000, 4000), device=devices[0])
x_gpu2 = torch.rand(size=(4000, 4000), device=devices[1])
```

```
devices = d2l.try_all_gpus()
def run(x):
    return [x.dot(x) for _ in range(50)]

x_gpu1 = np.random.uniform(size=(4000, 4000), ctx=devices[0])
x_gpu2 = np.random.uniform(size=(4000, 4000), ctx=devices[1])
```

```
[07:35:24] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
[07:35:25] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

次に、この関数をデータに適用する。キャッシュが結果に影響しないようにするため、計測の前にどちらか一方のデバイスで 1 回処理を行い、ウォームアップしておきる。torch.cuda.synchronize() は、CUDA デバイス上のすべてのストリームにあるすべてのカーネルの完了を待ちる。device 引数を受け取り、そのデバイスで同期が必要な場合に使う。device 引数が None（既定値）のときは、current_device() で与えられる現在のデバイスを使う。

```
run(x_gpu1)
run(x_gpu2)  # すべてのデバイスをウォームアップ
torch.cuda.synchronize(devices[0])
torch.cuda.synchronize(devices[1])

with d2l.Benchmark('GPU1 time'):
    run(x_gpu1)
    torch.cuda.synchronize(devices[0])

with d2l.Benchmark('GPU2 time'):
    run(x_gpu2)
    torch.cuda.synchronize(devices[1])
```

```
GPU1 time: 0.4469 sec
GPU2 time: 0.4567 sec
```

```
run(x_gpu1)  # 両方のデバイスをウォームアップ
run(x_gpu2)
```

(continues on next page)

(continued from previous page)

```
npx.waitall()

with d2l.Benchmark('GPU1 time'):
    run(x_gpu1)
    npx.waitall()

with d2l.Benchmark('GPU2 time'):
    run(x_gpu2)
    npx.waitall()
```

```
GPU1 time: 0.4586 sec
GPU2 time: 0.4602 sec
```

両方のタスクの間に synchronize 文を入れなければ、システムは両方のデバイス上での計算を自動的に並列化できる。

```
with d2l.Benchmark('GPU1 & GPU2'):
    run(x_gpu1)
    run(x_gpu2)
    torch.cuda.synchronize()
```

```
GPU1 & GPU2: 0.4467 sec
```

```
with d2l.Benchmark('GPU1 & GPU2'):
    run(x_gpu1)
    run(x_gpu2)
    npx.waitall()
```

```
GPU1 & GPU2: 0.4642 sec
```

上の例では、深層学習フレームワークがユーザ側で高度なコードを書く必要なく、両方の GPU デバイス上の計算を自動的にスケジューリングするため、総実行時間は各部分の合計より短くなる。

16.3.2 計算と通信の並列化

多くの場合、CPU と GPU の間、あるいは異なる GPU 間など、異なるデバイス間でデータを移動する必要がある。たとえば、複数のアクセラレータカード上で勾配を集約する必要がある分散最適化を行うときに、これが起こりる。これを、GPU 上で計算し、その結果を CPU にコピーすることでシミュレートしてみよう。

```
def copy_to_cpu(x, non_blocking=False):
    return [y.to('cpu', non_blocking=non_blocking) for y in x]
```

(continues on next page)

(continued from previous page)

```
with d2l.Benchmark('Run on GPU1'):
    y = run(x_gpu1)
    torch.cuda.synchronize()

with d2l.Benchmark('Copy to CPU'):
    y_cpu = copy_to_cpu(y)
    torch.cuda.synchronize()
```

```
Run on GPU1: 0.4480 sec
Copy to CPU: 2.2074 sec
```

```
def copy_to_cpu(x):
    return [y.copyto(npx.cpu()) for y in x]

with d2l.Benchmark('Run on GPU1'):
    y = run(x_gpu1)
    npx.waitall()

with d2l.Benchmark('Copy to CPU'):
    y_cpu = copy_to_cpu(y)
    npx.waitall()
```

```
Run on GPU1: 0.5041 sec
[07:35:27] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
Copy to CPU: 2.4251 sec
```

これはやや非効率である。y の一部を CPU にコピーし始めることは、リストの残りがまだ計算中であっても、すでに可能である。この状況は、たとえばミニバッチ上で（逆伝播による）勾配を計算するときに起こる。あるパラメータの勾配は、他のパラメータより先に得られる。したがって、GPU がまだ動作している間に PCI-Express バス帯域幅を使い始めることは有利である。PyTorch では、to() や copy_() などのいくつかの関数が明示的な non_blocking 引数を受け取り、不要な同期を呼び出し側が回避できる。non_blocking=True に設定することで、このシナリオをシミュレートできる。

```
with d2l.Benchmark('Run on GPU1 and copy to CPU'):
    y = run(x_gpu1)
    y_cpu = copy_to_cpu(y, True)
    torch.cuda.synchronize()
```

```
Run on GPU1 and copy to CPU: 1.7105 sec
```

```
with d2l.Benchmark('Run on GPU1 and copy to CPU'):
    y = run(x_gpu1)
```

(continues on next page)

(continued from previous page)

```
    y_cpu = copy_to_cpu(y)
    npx.waitall()
```

```
Run on GPU1 and copy to CPU: 0.4647 sec
```

両方の操作に必要な総時間は、予想どおり、それぞれの合計より短くなる。このタスクは別の資源、すなわち CPU と GPU の間のバスを使うため、並列計算とは異なることに注意しよう。実際には、両方のデバイスで計算しながら通信も同時に行える。上で述べたように、計算と通信の間には依存関係がある。`y[i]` は CPU にコピーされる前に計算されていなければならない。幸いなことに、システムは `y[i]` を計算しながら `y[i-1]` をコピーすることで、総実行時間を短縮できる。

最後に、Fig. 16.3.1 に示すように、CPU と 2 枚の GPU 上で学習する単純な 2 層 MLP における計算グラフとその依存関係を示す。これを手作業で並列プログラムとしてスケジューリングするのはかなり大変だろう。そこで、最適化のためにグラフベースの計算バックエンドを持つことが有利になる。

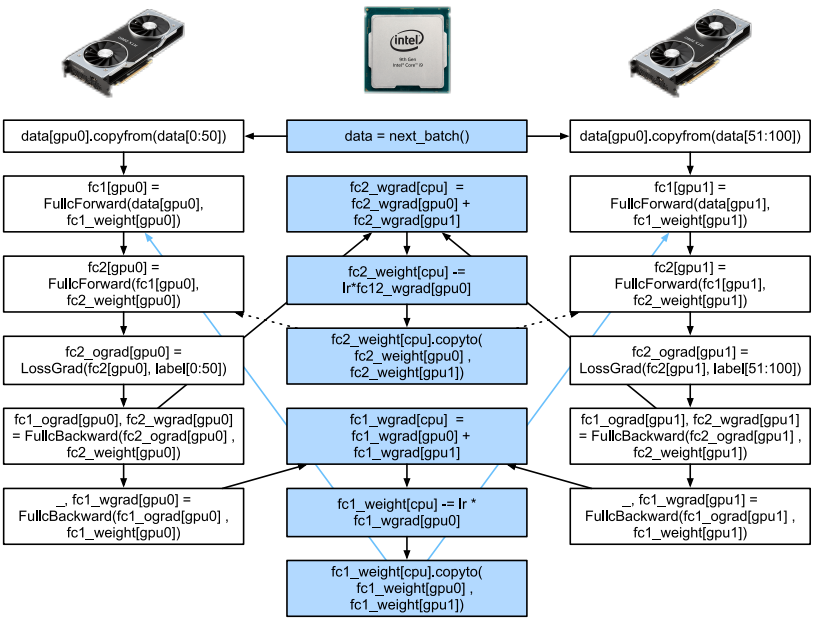


Fig. 16.3.1 2 層 MLP を CPU と 2 枚の GPU 上で実行したときの計算グラフとその依存関係。

16.3.3 要約

- 現代のシステムには、複数の GPU や CPU など、さまざまなデバイスがある。これらは非同期に並列利用できる。
- 現代のシステムには、PCI Express、ストレージ（通常はソリッドステートド

ライブ、またはネットワーク経由）、ネットワーク帯域幅など、通信のためのさまざまな資源もある。これらも最高効率のために並列利用できる。

- バックエンドは、自動並列計算と通信によって性能を向上できる。

16.3.4 演習

1. この節で定義した `run` 関数では 8 個の操作が実行された。これらの間に依存関係はない。深層学習フレームワークがそれらを自動的に並列実行するかどうかを調べる実験を設計せよ。
2. 個々の演算子のワークロードが十分に小さい場合、単一の CPU や GPU 上でも並列化が役立ちる。これを検証する実験を設計せよ。
3. CPU、GPU、および両デバイス間の通信を用いた並列計算の実験を設計せよ。
4. NVIDIA の Nsight[91] のようなデバッガを使って、コードが効率的であることを確認しなさい。
5. より複雑なデータ依存関係を含む計算タスクを設計し、性能を改善しつつ正しい結果が得られるかどうかを確認する実験を行いなさい。

16.4 ハードウェア

優れた性能を持つシステムを構築するには、問題の統計的側面を捉えるアルゴリズムやモデルをよく理解していることが必要である。同時に、基盤となるハードウェアについても少なくともある程度の知識を持っていることが不可欠である。本節は、ハードウェアやシステム設計の正式な講義の代わりにはならない。むしろ、なぜあるアルゴリズムが他より効率的なのか、そしてどうすれば高いスループットを達成できるのかを理解するための出発点として役立つだろう。優れた設計は容易に 1 桁の差を生み出し、その差が、ネットワークを訓練できるかどうか（たとえば 1 週間で）と、まったく訓練できないかどうか（3 か月かかって締切に間に合わない）を分けることさえある。

まずコンピュータを見る。次に、CPU と GPU をより詳しく見る。最後に、複数のコンピュータがサーバーセンターやクラウドでどのように接続されているかを概観する。

せっかちな読者は Fig. 16.4.1 だけでも十分かもしれない。これは Colin Scott の interactive post[92] から取ったもので、過去 10 年の進歩の概要をよく示している。元の数値は Jeff Dean の 2010 年の Stanford talk[93] に由来する。

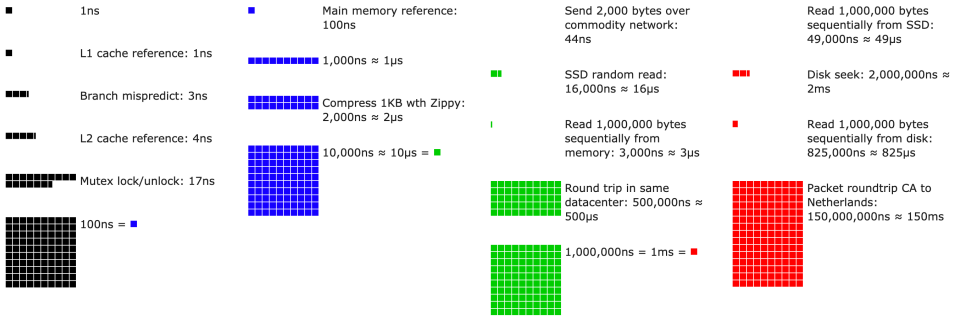


Fig. 16.4.1 Latency Numbers that every programmer should know.

以下の議論では、これらの数値の背後にあるいくつかの理由と、それらがアルゴリズム設計にどう役立つかを説明する。以下の議論は非常に高レベルで概略的なものである。明らかに、これは正式な講義の * 代わりにはならない *。むしろ、統計モデルを扱う人が適切な設計判断を下すのに十分な情報を提供することを目的としている。コンピュータアーキテクチャの詳細な概説については、(Hennessy and Patterson, 2011)、あるいは Arste Asanovic[94] によるような関連講義を参照しよ。

94 

16.4.1 コンピュータ

深層学習の研究者や実務者の多くは、かなりの量のメモリ、計算資源、GPU のような何らかのアクセラレータ、あるいはそれらを複数備えたコンピュータを利用できる。コンピュータは次の主要な構成要素から成る。

- プログラムを実行できるプロセッサ（CPU とも呼ばれる）。これはオペレーティングシステムや他の多くの処理も実行し、通常は 8 コア以上で構成される。
- 計算結果（重みベクトルや活性化、訓練データなど）を保存・取得するためのメモリ（RAM）。
- 1 GB/s から 100 GB/s の速度を持つ Ethernet ネットワーク接続（複数ある場合もある）。ハイエンドサーバーでは、より高度な相互接続が使われることもある。
- システムを 1 つ以上の GPU に接続するための高速拡張バス（PCIe）。サーバーでは最大 8 個のアクセラレータを備え、しばしば高度なトポロジーで接続される。一方、デスクトップシステムでは、予算や電源容量に応じて 1 個または 2 個である。
- 磁気ハードディスクドライブやソリッドステートドライブのような耐久性のあるストレージ。多くの場合 PCIe バスを使って接続される。これは、訓練データを効率よくシステムへ転送し、必要に応じて中間チェックポイントを保存する役割を果たする。

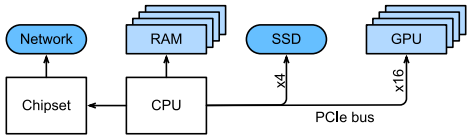


Fig. 16.4.2 Connectivity of components of a computer.

Fig. 16.4.2 が示すように、ほとんどの構成要素（ネットワーク、GPU、ストレージ）は PCIe バスを介して CPU に接続されている。PCIe は CPU に直接接続された複数のレーンから成る。たとえば AMD の Threadripper 3 は 64 本の PCIe 4.0 レーンを持ち、それぞれが双方向で 16 Gbit/s のデータ転送が可能です。メモリは CPU に直接接続され、総帯域幅は最大で 100 GB/s に達する。

コンピュータ上でコードを実行するときには、データをプロセッサ（CPU または GPU）へ送り、計算を行い、その結果をプロセッサから RAM や耐久性のあるストレージへ戻す必要がある。したがって、良い性能を得るには、どれか 1 つのシステムが大きなボトルネックにならないよう、これらが途切れなく連携することを　　する必要がある。たとえば、画像を十分な速さで読み込めなければ、プロセッサは仕事をすることができない。同様に、行列を CPU（または GPU）へ十分な速さで移動できなければ、その演算器は手待ちになる。最後に、ネットワークを介して複数のコンピュータを同期させたいなら、ネットワークが計算を遅らせてはならない。1 つの方法は、通信と計算を交互に行うことである。では、各構成要素をもう少し詳しく見ていこう。

16.4.2 メモリ

最も基本的には、メモリはすぐにアクセスできる必要のあるデータを保存するために使われる。現在、CPU の RAM は通常 DDR4[95] で、1 モジュールあたり 20–25 GB/s の帯域幅を提供する。各モジュールは 64 ビット幅のバスを持つ。通常は複数チャネルを使えるように、メモリモジュールは 2 枚組で使われる。CPU は 2〜4 のメモリチャネルを持ち、つまりピークメモリ帯域幅は 40 GB/s から 100 GB/s の範囲である。しばしば各チャネルに 2 つのバンクがある。たとえば AMD の Zen 3 Threadripper には 8 スロットある。

[95]

これらの数値は確かに印象的であるが、話の半分しか示していない。メモリの一部を読み出したいとき、まずその情報がどこにあるかをメモリモジュールに伝える必要がある。つまり、最初に RAM へアドレスを送らなければならない。これが済むと、64 ビットのレコードを 1 つだけ読むことも、長いレコード列を読むこともできる。後者は *burst read* と呼ばれる。要するに、メモリへアドレスを送り転送を開始するのに約 100 ns（詳細は使用するメモリチップのタイミング係数に依存する）かかり、その後の各転送はわずか 0.2 ns である。つまり、最初の読み出しは後続の読み出しの 500 倍も高価である！1 秒あたり最大 10,000,000 回のランダム読み出しが可能だとわかる。これは、できる限りランダムなメモリアクセスを避け、burst read（および write）を使うべきだことを示している。

複数の *bank* があることを考えると、事情は少し複雑になる。各 bank はかなり独立にメモリを読み出せる。これは 2 つの意味を持つ。

一方では、メモリ全体に均等に分散していれば、実効的なランダム読み出し回数は最大で 4 倍になる。また、burst read が 4 倍速い以上、ランダム読み出しを行うのは依然としてよくない、ということでもある。他方では、64 ビット境界へのメモリアラインメントのため、あらゆるデータ構造も同じ境界に合わせるのが望ましいである。コンパイラは適切なフラグが設定されていれば、これをかなり自動的に[96] 行う。興味のある読者は、Zeshan Chishti[97] による講義のような DRAM に関する講義を参照するとよいだろう。

96 

97 

GPU メモリは、CPU よりもはるかに多くの演算器を持つため、さらに高い帯域幅が要求される。これに対処する方法は大きく 2 つある。1 つ目は、メモリバスを大幅に広くすることである。たとえば NVIDIA の RTX 2080 Ti は 352 ビット幅のバスを持つ。これにより、より多くの情報を同時に転送できる。2 つ目は、GPU が特定の高性能メモリを使うことである。NVIDIA の RTX や Titan シリーズのような一般向けデバイスは通常、総帯域幅が 500 GB/s を超える GDDR6[98] チップを使う。別の選択肢として HBM（high bandwidth memory）モジュールがある。これは非常に異なるインターフェースを使い、専用のシリコンウェハー上で GPU に直接接続される。そのため非常に高価で、通常は NVIDIA Volta V100 シリーズのようなハイエンドサーバーチップに限って使われる。驚くことではないが、GPU メモリは CPU メモリよりも一般にはるかに小さいである。これは前者のコストが高いためである。ここでの目的に関しては、両者の性能特性は概ね似ているが、GPU のほうがずっと高速だと考えてよいだろう。本書の目的では、詳細は安全に無視できる。GPU カーネルを高スループット向けに調整するときにのみ重要になる。

98 

16.4.3 ストレージ

RAM の重要な特性として帯域幅とレイテンシがあることを見た。ストレージデバイスにも同じことが当てはまるが、その差はさらに極端になりえる。

ハードディスクドライブ

ハードディスクドライブ（HDD）は半世紀以上使われてきた。要するに、いくつかの回転するプラッタと、任意のトラックを読み書きするために位置決めできるヘッドから成る。ハイエンドのディスクは 9 枚のプラッタで最大 16 TB を保持する。HDD の大きな利点の 1 つは比較的安価であることである。多くの欠点の 1 つは、故障モードがしばしば壊滅的であり、読み出しレイテンシが比較的高いことである。

後者を理解するために、HDD が約 7,200 RPM（毎分回転数）で回転していることを考えてみよう。もしこれよりずっと速ければ、プラッタに働く遠心力のために破損してしまう。これは、ディスク上の特定のセクタへアクセスするときに大きな欠点になる。つまり、プラッタが所定の位置に回転してくるのを待た

なければならない（ヘッドは動かせるが、実際のディスクを加速することはできない）。そのため、要求したデータが利用可能になるまで 8 ms 以上かかることがある。これを表す一般的な言い方として、HDD は約 100 IOPs（input/output operations per second）で動作できる、というものがある。この数値は過去 20 年間ほとんど変わっていない。さらに悪いことに、帯域幅を増やすのも同様に難しいです（100–200 MB/s 程度です）。結局のところ、各ヘッドはビットのトラックを読み取るので、ビットレートは情報密度の平方根にしか比例して増えない。その結果、HDD は急速にアーカイブ用ストレージや、非常に大きなデータセット向けの低グレードなストレージへと追いやられつつある。

ソリッドステートドライブ

ソリッドステートドライブ（SSD）は、フラッシュメモリを使って情報を永続的に保存する。これにより、保存されたレコードへのアクセスがはるかに高速になる。現代の SSD は 100,000〜500,000 IOPs で動作でき、つまり HDD より最大で 3 桁高速である。さらに、帯域幅は 1–3GB/s に達し、HDD より 1 桁高速である。これらの改善は、うますぎて信じられないほどに聞こえる。実際、その設計方法に由来する次のような注意点がある。

- SSD は情報をブロック（256 KB 以上）単位で保存する。書き込みはブロック全体としてしか行えず、かなり時間がかかりる。したがって、SSD へのビット単位のランダム書き込みは非常に性能が悪いである。同様に、一般にデータを書き込むこと自体に時間がかかりる。なぜなら、ブロックを読み出し、消去し、その後新しい情報で書き直す必要があるからである。現在では SSD コントローラとファームウェアがこれを緩和するアルゴリズムを発達させている。それでも、特に QLC（quad level cell）SSD では書き込みがかなり遅くなりえる。性能向上の鍵は、操作のキューを維持し、読み出しを優先し、可能なら大きなブロックで書き込むことである。
- SSD のメモリセルは比較的早く劣化する（しばしば数千回の書き込み後にはすでに）。ウェアレベリング保護アルゴリズムは、劣化を多くのセルに分散できる。とはいえ、SSD をスワップファイルや大量のログファイルの集約に使うことは推奨されない。
- 最後に、帯域幅の大幅な増加により、コンピュータ設計者は SSD を PCIe バスに直接接続せざるを得なくなった。これに対応できるドライブは NVMe（Non Volatile Memory enhanced）と呼ばれ、最大 4 本の PCIe レーンを使える。PCIe 4.0 では最大 8GB/s に相当する。

クラウドストレージ

クラウドストレージは、設定可能な範囲の性能を提供する。つまり、仮想マシンへのストレージ割り当ては、量の面でも速度の面でも動的であり、ユーザーが選択できる。レイテンシが高すぎる場合、たとえば多数の小さなレコードを使った訓練中などには、割り当てられた IOP 数を増やすことを推奨する。

16.4.4 CPU

中央処理装置（CPU）は、あらゆるコンピュータの中心である。CPU は、機械語を実行できるプロセッサコア、それらを接続するバス（具体的なトポロジーはプロセッサのモデル、世代、ベンダーによって大きく異なる）、そして主記憶からの読み出しよりも高い帯域幅と低いレイテンシでメモリアクセスを可能にするキャッシュから成る。最後に、現代のほとんどすべての CPU は、メディア処理や機械学習で一般的な高性能線形代数や畳み込みを支援するためのベクトル処理ユニットを備えている。

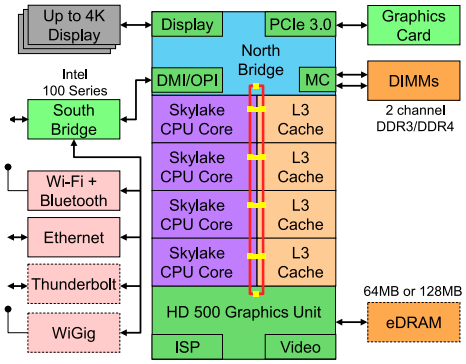


Fig. 16.4.3 Intel Skylake consumer quad-core CPU.

Fig. 16.4.3 は、Intel Skylake の一般向け 4 コア CPU を示している。これは統合 GPU、キャッシュ、4 コアを接続するリングバスを備えている。Ethernet、WiFi、Bluetooth、SSD コントローラ、USB のような周辺機器は、チップセットの一部であるか、あるいは CPU に直接（PCIe で）接続されている。

マイクロアーキテクチャ

各プロセッサコアは、かなり洗練された一連の構成要素から成る。詳細は世代やベンダーによって異なるが、基本機能はほぼ標準的である。フロントエンドは命令を読み込み、どの経路が選ばれるかを予測しようとする（たとえば制御フローに対して）。その後、命令はアセンブリコードからマイクロ命令へとデコードされる。アセンブリコードは、しばしばプロセッサが実行する最下層のコードではない。代わりに、複雑な命令はより低レベルの操作の集合へとデコードされることがある。これらは実際の実行コアによって処理される。後者はしばしば、多数の操作を同時に実行できる。たとえば、Fig. 16.4.4 の ARM Cortex A77 コアは、最大 8 つの操作を同時に実行できる。

これは、独立に実行できるなら、効率的なプログラムは 1 クロックサイクルあたり複数の命令を実行できる可能性があることを意味する。すべてのユニットが同じ能力を持つわけではない。整数命令を専門とするものもあれば、浮動小数点性能に最適化されたものもある。スループットを高めるため、プロセッサは分岐命令で複数のコード経路を同時に追跡し、実行されなかった分岐の結果

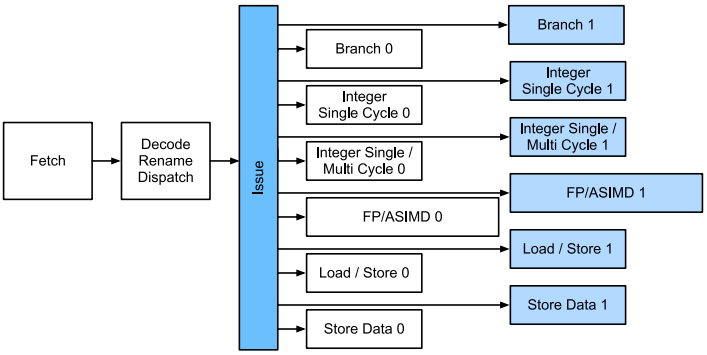


Fig. 16.4.4 ARM Cortex A77 Microarchitecture.

を破棄することもある。だからこそ、フロントエンドの分岐予測ユニットが重要であり、最も有望な経路だけが追跡されるのである。

ベクトル化

深層学習は非常に計算資源を消費する。したがって、CPU を機械学習に適したものにするには、1 クロックサイクルで多くの演算を行う必要がある。これはベクトルユニットによって実現される。名称はさまざまで、ARM では NEON、x86 では（最近の世代では）AVX2[99] ユニットと呼ばれる。共通しているのは、SIMD（single instruction multiple data）演算を実行できることである。Fig. 16.4.5 は、ARM で 8 個の短整数を 1 クロックサイクルで加算できる様子を示している。

99

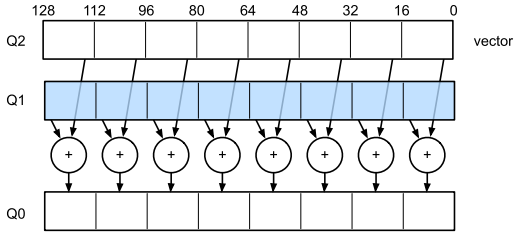


Fig. 16.4.5 128 bit NEON vectorization.

アーキテクチャの選択によっては、このようなレジスタは最大 512 ビット長で、最大 64 組の数値を組み合わせられる。たとえば、2 つの数を掛けて第 3 の数に加える、いわゆる fused multiply-add を行うことができる。Intel の OpenVino[100] は、これらを使ってサーバーグレード CPU 上で深層学習に十分なスループットを実現している。ただし、この数値は GPU が達成できる性能には完全に及ばない。たとえば、NVIDIA の RTX 2080 Ti は 4,352 個の CUDA コアを持ち、それぞれがそのような演算をいつでも処理できる。

100

キャッシュ

次の状況を考えてみよう。上で示した Fig. 16.4.3 のような、4 コアを持つ控えめな CPU コアが 2 GHz で動作しているとする。

さらに、IPC（instructions per clock）が 1 であり、256 ビット幅の AVX2 が有効になっていると仮定する。加えて、AVX2 演算に使うレジスタの少なくとも 1 つはメモリから取得する必要があるとする。これは、CPU が 1 クロックサイクルあたり 4×256 bit = 128 bytes のデータを消費することを意味する。1 秒あたり $2 \times 10^9 \times 128 = 256 \times 10^9$ バイトをプロセッサへ転送できなければ、演算器は手待ちになる。残念ながら、このようなチップのメモリインターフェースは 20–40 GB/s のデータ転送しかサポートせず、1 桁少ないのである。解決策は、できる限りメモリから新しいデータを読み込むのを避け、CPU 上に局所的にキャッシュすることである。そこでキャッシュが役立ちる。一般に次の名称や概念が使われる。

- **レジスタ**は厳密にはキャッシュの一部ではない。命令の段取りを助ける。とはいえ、CPU レジスタは CPU がクロック速度で遅延なくアクセスできるメモリ位置である。CPU には数十個のレジスタがある。レジスタを効率的に使うかどうかはコンパイラ（またはプログラマ）次第である。たとえば C 言語には `register` キーワードがある。

- **L1 キャッシュ**は、高いメモリ帯域幅要求に対する最初の防衛線である。L1 キャッシュは非常に小さく（典型的なサイズは 32–64 KB）、しばしばデータキャッシュと命令キャッシュに分かれている。データが L1 キャッシュに見つかれば、アクセスは非常に高速である。そこに見つからなければ、探索はキャッシュ階層の下位へ進みる。

- **L2 キャッシュ**が次の段階である。アーキテクチャ設計やプロセッササイズによっては排他的であることもある。特定のコアだけがアクセスできる場合もあれば、複数コアで共有される場合もある。L2 キャッシュは L1 より大きく（通常は 1 コアあたり 256–512 KB）、遅いである。さらに、L2 にあるものへアクセスするには、まず L1 にないことを確認する必要があり、その分わずかな追加レイテンシが生じる。

- **L3 キャッシュ**は複数コアで共有され、かなり大きくなりえる。AMD の Epyc 3 サーバー CPU は、複数のチップレットにまたがって合計 256 MB という巨大なキャッシュを持つ。より一般的な値は 4–8 MB 程度である。

次にどのメモリ要素が必要になるかを予測することは、チップ設計における重要な最適化パラメータの 1 つである。たとえば、多くのキャッシュアルゴリズムは後ろ向きよりも前向きに先読みしようとするため、メモリを前方方向に走査するのが望ましいである。同様に、メモリアクセスパターンを局所的に保つことは性能向上に有効である。

キャッシュを追加するのは諸刃の剣である。一方では、プロセッサコアがデータ不足に陥らないようにする。同時に、チップサイズを増やし、本来なら演算性能向上に使えたはずの面積を消費する。さらに、キャッシュミスは高くつく

ことがある。Fig. 16.4.6 に示すような最悪のケース、*false sharing* を考えてみよう。あるメモリ位置がプロセッサ 0 でキャッシュされているときに、プロセッサ 1 上のスレッドがそのデータを要求する。それを得るには、プロセッサ 0 は現在の処理を止め、情報を主記憶へ書き戻し、その後プロセッサ 1 がそれをメモリから読み出せるようにしなければならない。この操作中、両方のプロセッサは待機する。十分にありうることであるが、このようなコードは、効率的な単一プロセッサ実装と比べて、複数プロセッサ上では遅く動作する。これも、キャッシュサイズに実用上の限界がある理由の 1 つです（物理的なサイズ以外にも）。

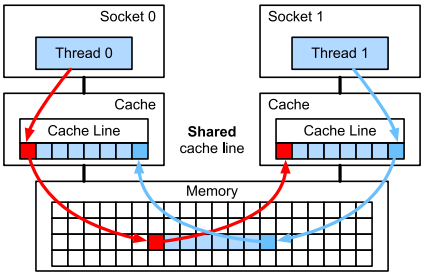


Fig. 16.4.6 False sharing (image courtesy of Intel).

16.4.5 GPU とその他のアクセラレータ

深層学習が GPU なしでは成功しなかったと主張しても大げさではない。同様に、GPU メーカーの業績が深層学習によって大きく向上したと論じるのも十分に妥当である。ハードウェアとアルゴリズムのこの共進化により、良くも悪くも深層学習が望ましい統計モデリングのパラダイムとなる状況が生まれた。したがって、GPU や TPU のような関連アクセラレータの具体的な利点を理解しておく価値がある (Jouppi *et al.*, 2017)。

実務でしばしば行われる区別として、アクセラレータは訓練向けか推論向けかのどちらかに最適化される。推論では、ネットワークの順伝播を計算するだけで十分である。逆伝播のための中間データ保存は不要である。さらに、非常に高精度な計算も必要ないかもしれない（通常は FP16 や INT8 で十分です）。一方、訓練では勾配を計算するためにすべての中間結果を保存する必要がある。さらに、勾配の蓄積には数値のアンダーフロー（またはオーバーフロー）を避けるため、より高い精度が必要である。つまり、FP16（または FP32 との混合精度）が最低要件になる。これらすべてにより、より高速で大容量のメモリ（HBM2 対 GDDR6）と、より多くの演算性能が必要になる。たとえば、NVIDIA の Turing[101] T4 GPU は推論向けに最適化されており、一方 V100 GPU は訓練により適している。

101

Fig. 16.4.5 で示したベクトル化を思い出してほしい。プロセッサコアにベクトルユニットを追加することで、スループットを大幅に向上させることができた。たとえば、Fig. 16.4.5 の例では 16 個の演算を同時に実行できた。

まず、ベクトル間の演算だけでなく行列間の演算も最適化するような演算を追加したらどうだろうか。この戦略が tensor core につながった（まもなく説明する）。

次に、もっと多くのコアを追加したらどうだろうか。要するに、この 2 つの戦略が GPU の設計判断を要約している。Fig. 16.4.7 は基本的な処理ブロックの概要を示している。そこには 16 個の整数演算ユニットと 16 個の浮動小数点演算ユニットがある。それに加えて、2 つの tensor core が、深層学習に関連する追加演算のうち狭い範囲を高速化する。各 streaming multiprocessor は、そのようなブロックを 4 つ含む。

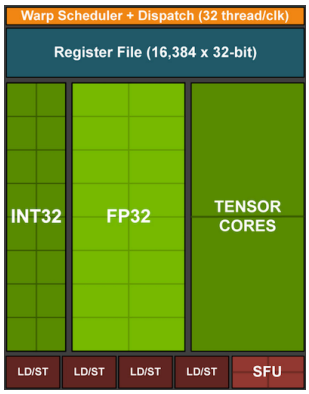


Fig. 16.4.7 NVIDIA Turing processing block (image courtesy of NVIDIA).

次に、12 個の streaming multiprocessor が graphics processing cluster にまとめられ、それがハイエンドの TU102 プロセッサを構成する。十分なメモリチャネルと L2 キャッシュがこの構成を補完する。Fig. 16.4.8 に関連する詳細がある。このようなデバイスを設計する理由の 1 つは、必要に応じて個々のブロックを追加・削除でき、よりコンパクトなチップを実現したり、歩留まりの問題（不良モジュールが有効化されない場合など）に対処したりできることである。幸いなことに、こうしたデバイスのプログラミングは、CUDA やフレームワークコードの層の下に隠れており、一般的な深層学習研究者が直接意識することはほとんどない。特に、利用可能な資源があれば、複数のプログラムが GPU 上で同時に実行されることも十分ありえる。それでも、デバイスメモリに収まらないモデルを選ばないよう、デバイスの制約を知っておく価値はある。

最後に、より詳しく触れておく価値があるのは *tensor core* である。これは、深層学習に特に有効な、より最適化された回路を追加する最近の傾向の一例である。たとえば、TPU は高速な行列乗算のために systolic array (Kung, 1988) を追加した。そこでは、非常に少数（第 1 世代 TPU では 1 つ）の大きな演算をサポートする設計でした。tensor core はその反対側にある。数値精度に応じて 4×4 から 16×16 の行列を扱う小さな演算に最適化されている。Fig. 16.4.9 は最適化の概要を示している。

もちろん、計算を最適化すると、ある種の妥協は避けられない。その 1 つは、



Fig. 16.4.8 NVIDIA Turing architecture (image courtesy of NVIDIA)

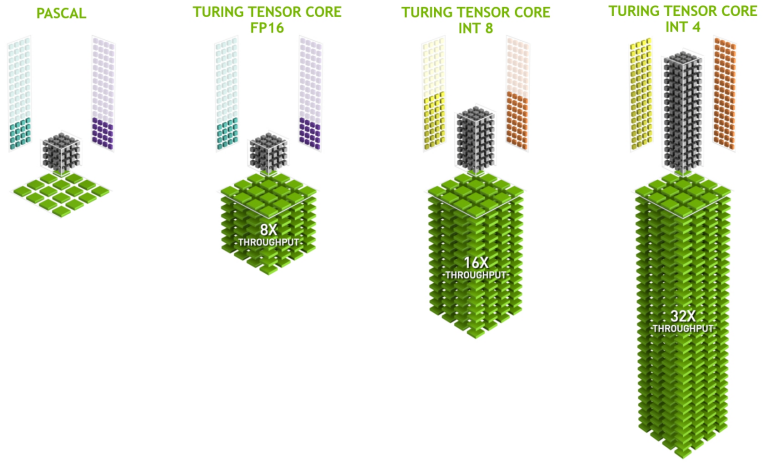


Fig. 16.4.9 NVIDIA tensor cores in Turing (image courtesy of NVIDIA).

GPU は割り込みや疎なデータの処理があまり得意ではないことである。Gunrock [102] (Wang *et al.*, 2016) のような注目すべき例外はあるが、疎行列や疎ベクトルのアクセスパターンは、GPU が得意とする高帯域幅の burst read 操作とは相性がよくない。両方の目標を両立させることは活発な研究分野である。たとえば、グラフ上の深層学習向けに調整されたライブラリ DGL [103] を参照しよ。

102

103

16.4.6 ネットワークとバス

単一のデバイスだけでは最適化に不十分な場合、処理を同期するためにデータをそのデバイスへ、またはそこから転送する必要がある。そこでネットワークとバスが役立ちる。設計パラメータとしては、帯域幅、コスト、距離、柔軟性がある。

一方には WiFi がある。これはかなり良い到達範囲を持ち、使うのが非常に簡単で（何しろ配線が不要です）、安価であるが、帯域幅とレイテンシは比較的平凡である。正気な機械学習研究者なら、これでサーバークラスタを構築しようとはしないだろう。以下では、深層学習に適した相互接続に焦点を当てる。

- **PCIe** は、非常に高帯域幅のポイントツーポイント接続のための専用バスです（PCIe 4.0 の 16 レーンスロットでは、1 レーンあたり最大 32 GB/s）。レイテンシは 1 桁マイクロ秒（5 µs）程度である。PCIe リンクは貴重である。プロセッサが持つ数は限られている。AMD の EPYC 3 は 128 レーン、Intel の Xeon は 1 チップあたり最大 48 レーンである。デスクトップ向け CPU では、それぞれ 20（Ryzen 9）と 16（Core i9）である。GPU は通常 16 レーンを使うため、CPU にフル帯域で接続できる GPU の数は制限される。結局のところ、ストレージや Ethernet のような他の高帯域幅周辺機器とリンクを共有しなければならないからである。RAM アクセスと同様に、パケットのオーバーヘッドが減るため、大きな一括転送が望ましいである。

- **Ethernet** は、コンピュータを接続する最も一般的な方法である。PCIe よりかなり遅いものの、導入コストが非常に低く、堅牢で、はるかに長い距離をカバーできる。低グレードのサーバーでは、典型的な帯域幅は 1 GBit/s である。より高性能なデバイス（たとえばクラウドの C5 instances[104] ）では、10〜100 GBit/s の帯域幅が提供される。これまでと同様に、データ伝送にはかなりのオーバーヘッドがある。生の Ethernet を直接使うことはほとんどなく、物理的な相互接続の上で実行されるプロトコル（UDP や TCP/IP など）を使う。これにより、さらにオーバーヘッドが増える。PCIe と同様に、Ethernet は 2 つのデバイス、たとえばコンピュータとスイッチを接続するよう設計されている。

104

- **スイッチ**を使うと、任意の 2 台が（通常はフル帯域幅の）ポイントツーポイント接続を同時に行える形で、複数のデバイスを接続できる。たとえば、Ethernet スイッチは 40 台のサーバーを高い断面帯域幅で接続できる。スイッチは従来のコンピュータネットワークに固有のものではない。PCIe レーンでさえスイッチング[105] できる。これは、たとえば P2 instances[106] のように、多数の GPU をホストプロセッサに接続する場合に起こりる。

105

- **NVLink** は、非常に高帯域幅の相互接続における PCIe の代替である。1 リンクあたり最大 300 Gbit/s のデータ転送速度を提供する。サーバー向け GPU（Volta V100）は 6 リンクを持つが、一般向け GPU（RTX 2080 Ti）は 1 リンクしかなく、速度も 100 Gbit/s に抑えられている。GPU 間で高いデータ転送性能を得るには NCCL[107] の使用を推奨する。

106

107

16.4.7 さらなるレイテンシ数値

Table 16.4.1 と Table 16.4.2 の要約は、数値の更新版を GitHub gist[108] として管理している Eliot Eshelman[109] によるものです。

108



109



Table 16.4.1: Common Latency Numbers.

Action	Time	Notes
L1 cache reference/hit	1.5 ns	4 cycles
Floating-point add/mult/FMA	1.5 ns	4 cycles
L2 cache reference/hit	5 ns	12 ~ 17 cycles
Branch mispredict	6 ns	15 ~ 20 cycles
L3 cache hit (unshared cache)	16 ns	42 cycles
L3 cache hit (shared in another core)	25 ns	65 cycles
Mutex lock/unlock	25 ns	
L3 cache hit (modified in another core)	29 ns	75 cycles
L3 cache hit (on a remote CPU socket)	40 ns	100 ~ 300 cycles (40 ~ 116 ns)
QPI hop to a another CPU (per hop)	40 ns	
64MB memory ref. (local CPU)	46 ns	TinyMemBench on Broadwell E5-2690v4
64MB memory ref. (remote CPU)	70 ns	TinyMemBench on Broadwell E5-2690v4
256MB memory ref. (local CPU)	75 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane random write	94 ns	UCSD Non-Volatile Systems Lab
256MB memory ref. (remote CPU)	120 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane random read	305 ns	UCSD Non-Volatile Systems Lab
Send 4KB over 100 Gbps HPC fabric	1 µs	MVAPICH2 over Intel Omni-Path
Compress 1KB with Google Snappy	3 µs	
Send 4KB over 10 Gbps ethernet	10 µs	
Write 4KB randomly to NVMe SSD	30 µs	DC P3608 NVMe SSD (QOS 99% is 500µs)
Transfer 1MB to/from NVLink GPU	30 µs	~33GB/s on NVIDIA 40GB NVLink
Transfer 1MB to/from PCI-E GPU	80 µs	~12GB/s on PCIe 3.0 x16 link
Read 4KB randomly from NVMe SSD	120 µs	DC P3608 NVMe SSD (QOS 99%)
Read 1MB sequentially from NVMe SSD	208 µs	~4.8GB/s DC P3608 NVMe SSD
Write 4KB randomly to SATA SSD	500 µs	DC S3510 SATA SSD (QOS 99.9%)
Read 4KB randomly from SATA SSD	500 µs	DC S3510 SATA SSD (QOS 99.9%)
Round trip within same data center	500 µs	One-way ping is ~250µs
Read 1MB sequentially from SATA SSD	2 ms	~550MB/s DC S3510 SATA SSD
Read 1MB sequentially from disk	5 ms	~200MB/s server HDD
Random Disk Access (seek+rotation)	10 ms	
Send packet CA->Netherlands->CA	150 ms	

Table 16.4.2: Latency Numbers for NVIDIA Tesla GPUs.

Action	Time	Notes
GPU Shared Memory access	30 ns	30~90 cycles (bank conflicts add latency)
GPU Global Memory access	200 ns	200~800 cycles
Launch CUDA kernel on GPU	10 µs	Host CPU instructs GPU to start kernel
Transfer 1MB to/from NVLink GPU	30 µs	~33GB/s on NVIDIA 40GB NVLink
Transfer 1MB to/from PCI-E GPU	80 µs	~12GB/s on PCI-Express x16 link

16.4.8 まとめ

- デバイスには操作ごとのオーバーヘッドがある。したがって、多数の小さな転送よりも、少数の大きな転送を目指すことが重要である。これは RAM、SSD、ネットワーク、GPU に当てはまる。
- 性能にはベクトル化が重要である。自分のアクセラレータが持つ具体的な能力を把握しておきよう。たとえば、ある Intel Xeon CPU は INT8 演算に特に優れ、NVIDIA Volta GPU は FP16 の行列-行列演算で卓越し、NVIDIA Turing は FP16、INT8、INT4 演算で優れている。
- 小さなデータ型による数値オーバーフローは、訓練中（推論ではそれよりは少ないですが）問題になることがある。
- エイリアシングは性能を大きく低下させることがある。たとえば、64 ビット CPU でのメモリアラインメントは 64 ビット境界に合わせるべきである。GPU では、畳み込みサイズを tensor core に合わせて整列させるのがよいだろう。
- アルゴリズムをハードウェアに合わせましょう（たとえばメモリフットプリントや帯域幅）。パラメータがキャッシュに収まると、非常に大きな高速化（1 桁以上）が得られる。
- 新しいアルゴリズムの性能は、実験結果を検証する前に紙の上で見積もることを推奨する。1 桁以上の不一致は懸念材料である。
- 性能ボトルネックのデバッグにはプロファイラを使いよう。
- 訓練用と推論用のハードウェアは、価格性能比の観点で異なる最適点を持つ。

16.4.9 演習

1. 外部メモリインターフェースに対して整列している場合と不整列な場合で、メモリアクセス速度に差があるかどうかを調べる C コードを書きなさい。ヒント: キャッシュ効果に注意すること。
2. 連続アクセスと、あるストライドを持つアクセスの速度差を調べなさい。

3. CPU のキャッシュサイズをどのように測定できるか？
4. 最大帯域幅を得るには、複数のメモリチャネルにデータをどのように配置するか？多数の小さなスレッドがある場合はどうするか？
5. エンタープライズクラスの HDD が 10,000 rpm で回転しているとする。最悪の場合、HDD がデータを読み出せるようになるまでに絶対に必要な最小時間はどれくらいか（ヘッドはほぼ瞬時に動くと仮定してよい）？なぜ 2.5”HDD が商用サーバーで人気になりつつあるのだろうか。（3.5”や 5.25”ドライブと比べて）？
6. HDD メーカーが記録密度を 1 平方インチあたり 1 Tbit から 5 Tbit に増やしたとする。2.5”HDD の 1 つのトラック上にどれだけの情報を保存できるか？内側トラックと外側トラックで違いはあるか？
7. 8 ビットから 16 ビットのデータ型にすると、必要なシリコン量はおよそ 4 倍になる。なぜだろうか。NVIDIA が Turing GPU に INT4 演算を追加したのはなぜだと思うか？
8. メモリを前方に読むのと後方に読むのでは、どれくらい速さが違いますか？この数値はコンピュータや CPU ベンダーによって異なるか？なぜだろうか。C コードを書いて実験しなさい。
9. ディスクのキャッシュサイズを測定できるか？一般的な HDD ではどれくらいですか？SSD にもキャッシュは必要ですか？
10. Ethernet を介してメッセージを送るときのパケットオーバーヘッドを測定しなさい。UDP と TCP/IP 接続の違いを調べなさい。
11. Direct memory access により、CPU 以外のデバイスがメモリへ直接書き込み（および読み出し）できる。なぜこれは良い考えなのだろうか。
12. Turing T4 GPU の性能数値を見なさい。FP16 から INT8 や INT4 へ移ると性能が「たった」2 倍にしかならないのはなぜだろうか。
13. サンフランシスコとアムステルダムの間で、パケットの往復にかかる最短時間はどれくらいであるべきですか？ヒント: 距離は 10,000 km と仮定してよいである。

16.5 複数 GPU での学習

これまで、CPU と GPU 上でモデルを効率よく学習する方法について議論してきた。さらに、Section 16.3 では、深層学習フレームワークがそれらの間で計算と通信を自動的に並列化できることも示した。また、Section 9.7 では、`nvidia-smi` コマンドを使ってコンピュータ上で利用可能な GPU をすべて一覧表示する方法も示した。

しかし、実際に深層学習の学習をどのように並列化するのかについては、まだ議論していなかった。
その代わりに、データを何らかの形で複数デバイスに分割して動作させる、ということを暗黙に仄めかしていた。本節ではその詳細を補い、ゼロからネットワークを並列に学習する方法を示す。高レベル API の機能を活用する方法については、Section 16.6 に譲る。
ここでは、Section 15.5 で説明したようなミニバッチ確率的勾配降下法に慣れていることを前提とする。

16.5.1 問題の分割

まずは、単純なコンピュータビジョン問題と、やや古典的なネットワーク、たとえば複数層の畳み込み、プーリング、そして最後にいくつかの全結合層を持つものから始めよう。
つまり、LeNet (LeCun *et al.*, 1998) や AlexNet (Krizhevsky *et al.*, 2012) にかなり似たネットワークから始める。
複数の GPU（デスクトップサーバなら 2 枚、AWS g4dn.12xlarge インスタンスなら 4 枚、p3.16xlarge なら 8 枚、p2.16xlarge なら 16 枚）があるとき、単純で再現性のある設計方針の恩恵を受けつつ、良好な高速化を達成できるように学習を分割したいと考える。結局のところ、複数 GPU は メモリと 計算能力の両方を増やする。要するに、分類したい学習データのミニバッチが与えられたとき、次の選択肢がある。

まず、ネットワークを複数 GPU にまたがって分割する方法がある。つまり、各 GPU が特定の層へ流れ込むデータを入力として受け取り、いくつかの後続層にわたってデータを処理し、その後データを次の GPU へ送りる。
これにより、単一 GPU では扱えないより大きなネットワークでデータを処理できる。
さらに、GPU ごとのメモリ使用量も十分に制御できる（全ネットワークのメモリ使用量の一部で済みます）。

しかし、層間のインターフェース（したがって GPU 間のインターフェース）には厳密な同期が必要である。特に、層ごとの計算負荷が適切に釣り合っていない場合、これは厄介である。GPU の数が増えると、この問題はさらに悪化する。
層間のインターフェースでは、活性化や勾配など、大量のデータ転送も必要になる。
これは GPU バスの帯域幅を圧迫する可能性がある。
さらに、計算集約的でありながら逐次的な操作を分割するのは容易ではない。この点についての最善の試みとしては、たとえば Mirhoseini *et al.* (2017) を参照

しよ。これは依然として難しい問題であり、非自明な問題に対して良い（線形な）スケーリングを達成できるかどうかは明らかではない。複数 GPU を連結するための優れたフレームワークやオペレーティングシステムの支援がない限り、推奨しない。

第二に、作業を層単位で分割する方法がある。たとえば、単一 GPU で 64 チャネルを計算する代わりに、問題を 4 枚の GPU に分割し、それぞれが 16 チャネル分のデータを生成するようにできる。

同様に、全結合層では出力ユニット数を分割できる。Fig. 16.5.1（Krizhevsky *et al.* (2012) より引用）はこの設計を示しており、当時は GPU のメモリ容量が非常に小さかった（当時 2GB）ため、この戦略が用いられていた。

チャネル数（またはユニット数）が小さすぎない限り、計算の観点では良好なスケーリングが可能である。

さらに、利用可能メモリは線形に増えるため、複数 GPU を使えばより大きなネットワークを処理できる。

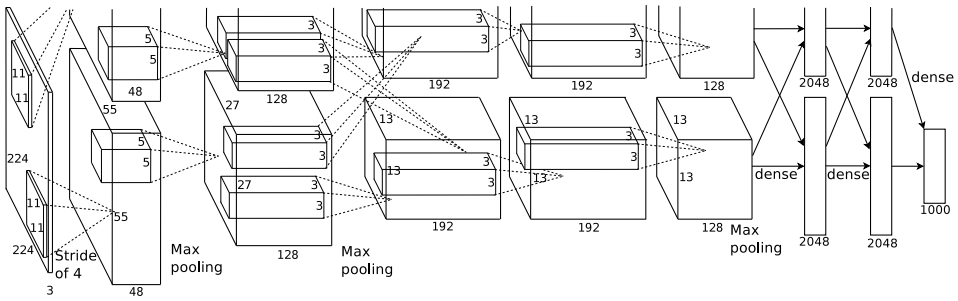


Fig. 16.5.1 メモリ制約のために元の AlexNet 設計で用いられたモデル並列。

しかし、各層が他のすべての層の結果に依存するため、同期操作やバリア操作が 非常に多く必要になる。

さらに、転送が必要なデータ量は、GPU 間で層を分散する場合よりもさらに大きくなる可能性がある。したがって、帯域幅コストと複雑さのため、この方法は推奨しない。

最後に、データを複数 GPU に分割する方法がある。この方法では、すべての GPU が異なる観測に対してではあるものの、同じ種類の作業を行う。各ミニバッチの学習後に、勾配が GPU 間で集約される。

これは最も単純な方法であり、どのような状況にも適用できる。

必要なのは、各ミニバッチの後に同期することだけである。とはいえ、他の勾

配がまだ計算中のうちから、勾配パラメータの交換を始められると非常に望ましいである。
さらに、GPU 数が増えるとミニバッチサイズも大きくなるため、学習効率が向上する。
ただし、GPU を増やしても、より大きなモデルを学習できるようにはならない。

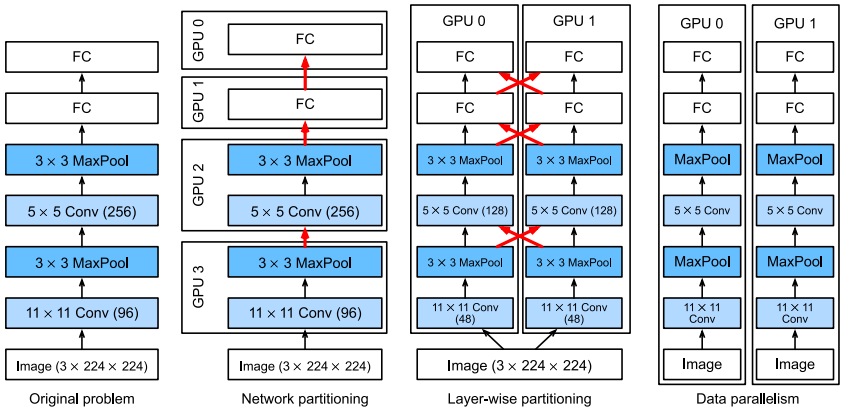


Fig. 16.5.2 複数 GPU 上での並列化。左から右へ：元の問題、ネットワーク分割、層単位分割、データ並列。

複数 GPU でのさまざまな並列化方法の比較を Fig. 16.5.2 に示す。
概して、十分に大きなメモリを持つ GPU にアクセスできるなら、データ並列が最も便利な方法である。分散学習のための分割についての詳細な説明は (Li *et al.*, 2014) も参照しよ。深層学習初期には GPU メモリが問題でしたが、現在では非常に特殊な場合を除いてこの問題は解決されている。以下ではデータ並列に焦点を当てる。

16.5.2 データ並列

1 台のマシンに k 枚の GPU があると仮定する。学習するモデルが与えられると、各 GPU は独立にモデルパラメータの完全な集合を保持する。ただし、GPU 間のパラメータ値は同一であり、同期されている。
例として、Fig. 16.5.3 は $k = 2$ のときのデータ並列による学習を示している。

一般に、学習は次のように進みる。

- 学習の任意の反復で、ランダムなミニバッチが与えられたら、バッチ内の例を k 個の部分に分け、GPU に均等に分配する。

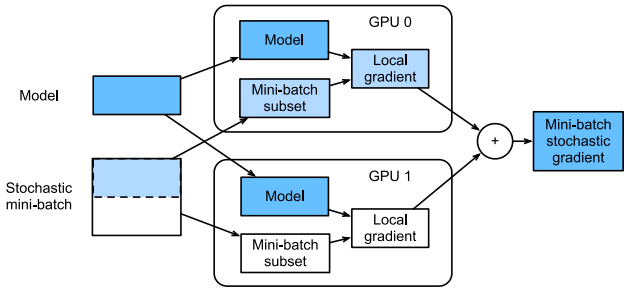


Fig. 16.5.3 2 枚の GPU 上でデータ並列を用いてミニバッチ確率的勾配降下法を計算する様子。

- 各 GPU は、自分に割り当てられたミニバッチ部分に基づいて、モデルパラメータの損失と勾配を計算する。
- k 枚の GPU の局所勾配を集約して、現在のミニバッチ確率的勾配を得る。
- 集約された勾配を各 GPU に再配布する。
- 各 GPU は、このミニバッチ確率的勾配を使って、自分が保持しているモデルパラメータの完全な集合を更新する。

実際には、k 枚の GPU で学習するときにはミニバッチサイズを k 倍に 増やすのが普通である。そうすることで、各 GPU が単一 GPU で学習する場合と同じ量の仕事をすることになる。16 枚 GPU のサーバでは、ミニバッチサイズがかなり大きくなるため、それに応じて学習率も増やす必要があるかもしれない。

また、Section 11.5 のバッチ正規化は調整が必要で、たとえば GPU ごとに別々のバッチ正規化係数を保持する方法がある。

以下では、複数 GPU 学習を説明するためにおもちゃのネットワークを使う。

```
%matplotlib inline
from d2l import torch as d2l
import torch
from torch import nn
from torch.nn import functional as F
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, np, npx
npx.set_np()
```

```
new_params = get_params(params, d2l.try_gpu(0))
print('b1 weight:', new_params[1])
print('b1 grad:', new_params[1].grad)
```

```
new_params = get_params(params, d2l.try_gpu(0))
print('b1 weight:', new_params[1])
print('b1 grad:', new_params[1].grad)
```

16.5.3 おもちゃのネットワーク

Section 10.6 で導入した LeNet を少し変更して使う。パラメータ交換と同期を詳細に示すため、これをゼロから定義する。

```
# Initialize model parameters
scale = 0.01
W1 = torch.randn(size=(20, 1, 3, 3)) * scale
b1 = torch.zeros(20)
W2 = torch.randn(size=(50, 20, 5, 5)) * scale
b2 = torch.zeros(50)
W3 = torch.randn(size=(800, 128)) * scale
b3 = torch.zeros(128)
W4 = torch.randn(size=(128, 10)) * scale
b4 = torch.zeros(10)
params = [W1, b1, W2, b2, W3, b3, W4, b4]

# Define the model
def lenet(X, params):
    h1_conv = F.conv2d(input=X, weight=params[0], bias=params[1])
    h1_activation = F.relu(h1_conv)
    h1 = F.avg_pool2d(input=h1_activation, kernel_size=(2, 2), stride=(2, 2))
    h2_conv = F.conv2d(input=h1, weight=params[2], bias=params[3])
    h2_activation = F.relu(h2_conv)
    h2 = F.avg_pool2d(input=h2_activation, kernel_size=(2, 2), stride=(2, 2))
    h2 = h2.reshape(h2.shape[0], -1)
    h3_linear = torch.mm(h2, params[4]) + params[5]
    h3 = F.relu(h3_linear)
    y_hat = torch.mm(h3, params[6]) + params[7]
    return y_hat

# Cross-entropy loss function
loss = nn.CrossEntropyLoss(reduction='none')
```

```
# Initialize model parameters
scale = 0.01
W1 = np.random.normal(scale=scale, size=(20, 1, 3, 3))
b1 = np.zeros(20)
W2 = np.random.normal(scale=scale, size=(50, 20, 5, 5))
b2 = np.zeros(50)
W3 = np.random.normal(scale=scale, size=(800, 128))
b3 = np.zeros(128)
W4 = np.random.normal(scale=scale, size=(128, 10))
b4 = np.zeros(10)
params = [W1, b1, W2, b2, W3, b3, W4, b4]

# Define the model
def lenet(X, params):
```

(continues on next page)

(continued from previous page)

```
    h1_conv = npx.convolution(data=X, weight=params[0], bias=params[1],
                              kernel=(3, 3), num_filter=20)
    h1_activation = npx.relu(h1_conv)
    h1 = npx.pooling(data=h1_activation, pool_type='avg', kernel=(2, 2),
                     stride=(2, 2))
    h2_conv = npx.convolution(data=h1, weight=params[2], bias=params[3],
                              kernel=(5, 5), num_filter=50)
    h2_activation = npx.relu(h2_conv)
    h2 = npx.pooling(data=h2_activation, pool_type='avg', kernel=(2, 2),
                     stride=(2, 2))
    h2 = h2.reshape(h2.shape[0], -1)
    h3_linear = np.dot(h2, params[4]) + params[5]
    h3 = npx.relu(h3_linear)
    y_hat = np.dot(h3, params[6]) + params[7]
    return y_hat

# Cross-entropy loss function
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

```
[07:25:19] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
train(num_gpus=1, batch_size=256, lr=0.2)
```

```
train(num_gpus=1, batch_size=256, lr=0.2)
```

16.5.4 データ同期

効率的な複数 GPU 学習には、2 つの基本操作が必要である。
まず、パラメータのリストを複数デバイスに配布する能力と、勾配を付加する能力（get_params）が必要である。パラメータがなければ、GPU 上でネットワークを評価することはできない。
第二に、複数デバイス間でパラメータを合計する能力、つまり allreduce 関数が必要である。

```
def get_params(params, device):
    new_params = [p.to(device) for p in params]
    for p in new_params:
        p.requires_grad_()
    return new_params
```

```
def get_params(params, device):
    new_params = [p.copyto(device) for p in params]
    for p in new_params:
```

(continues on next page)

(continued from previous page)

```
        p.attach_grad()
    return new_params
```

```
train(num_gpus=2, batch_size=256, lr=0.2)
```


```
train(num_gpus=2, batch_size=256, lr=0.2)
```


モデルパラメータを 1 枚の GPU にコピーして試してみよう。

```
new_params = get_params(params, d2l.try_gpu(0))
print('b1 weight:', new_params[1])
print('b1 grad:', new_params[1].grad)
```

```
b1 weight: tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,␣
↪0., 0., 0., 0., 0.],
       device='cuda:0', requires_grad=True)
b1 grad: None
```

まだ何の計算もしていないので、バイアスパラメータに関する勾配はまだゼロである。
次に、複数 GPU に分散されたベクトルがあると仮定しよう。次の `allreduce` 関数は、すべてのベクトルを加算し、その結果をすべての GPU にブロードキャストする。これを動かすには、結果を集約するデバイスへデータをコピーする必要があることに注意しよう。

```
def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].to(data[0].device)
    for i in range(1, len(data)):
        data[i][:] = data[0].to(data[i].device)
```

```
def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].copyto(data[0].ctx)
    for i in range(1, len(data)):
        data[0].copyto(data[i])
```

異なるデバイス上で異なる値を持つベクトルを作成し、それらを集約してみよう。

```
data = [torch.ones((1, 2), device=d2l.try_gpu(i)) * (i + 1) for i in range(2)]
print('before allreduce:\n', data[0], '\n', data[1])
allreduce(data)
print('after allreduce:\n', data[0], '\n', data[1])
```

```
before allreduce:
 tensor([[1., 1.]], device='cuda:0')
 tensor([[2., 2.]], device='cuda:1')
after allreduce:
 tensor([[3., 3.]], device='cuda:0')
 tensor([[3., 3.]], device='cuda:1')
```

```
data = [np.ones((1, 2), ctx=d2l.try_gpu(i)) * (i + 1) for i in range(2)]
print('before allreduce:\n', data[0], '\n', data[1])
allreduce(data)
print('after allreduce:\n', data[0], '\n', data[1])
```

```
[07:25:20] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
before allreduce:
 [[1. 1.]] @gpu(0)
 [[2. 2.]] @gpu(1)
after allreduce:
 [[3. 3.]] @gpu(0)
 [[3. 3.]] @gpu(1)
```

16.5.5 データの分配

ミニバッチを複数 GPU に均等に分配するための簡単なユーティリティ関数が必要である。たとえば、2 枚の GPU では、データの半分をそれぞれの GPU にコピーしたいとする。
より便利で簡潔なので、まずは深層学習フレームワークの組み込み関数を使って、4 × 5 行列で試してみよう。

```
data = torch.arange(20).reshape(4, 5)
devices = [torch.device('cuda:0'), torch.device('cuda:1')]
split = nn.parallel.scatter(data, devices)
print('input :', data)
print('load into', devices)
print('output:', split)
```

```
input : tensor([[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]])
```

(continues on next page)

(continued from previous page)

```
load into [device(type='cuda', index=0), device(type='cuda', index=1)]
output: (tensor([[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]], device='cuda:0'), tensor([[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]], device='cuda:1'))
```

```
data = np.arange(20).reshape(4, 5)
devices = [npx.gpu(0), npx.gpu(1)]
split = gluon.utils.split_and_load(data, devices)
print('input :', data)
print('load into', devices)
print('output:', split)
```

```
input : [[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14.]
 [15. 16. 17. 18. 19.]]
load into [gpu(0), gpu(1)]
output: [array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]], ctx=gpu(0)), array([[10., 11., 12., 13., 14.],
       [15., 16., 17., 18., 19.]], ctx=gpu(1))]
```

後で再利用できるように、データとラベルの両方を分割する split_batch 関数を定義する。

```
#@save
def split_batch(X, y, devices):
    """Split `X` and `y` into multiple devices."""
    assert X.shape[0] == y.shape[0]
    return (nn.parallel.scatter(X, devices),
            nn.parallel.scatter(y, devices))
```

```
#@save
def split_batch(X, y, devices):
    """Split `X` and `y` into multiple devices."""
    assert X.shape[0] == y.shape[0]
    return (gluon.utils.split_and_load(X, devices),
            gluon.utils.split_and_load(y, devices))
```

16.5.6 学習

これで、単一ミニバッチに対する複数 GPU 学習を実装できる。その実装は主として、本節で説明したデータ並列のアプローチに基づいている。複数 GPU 間でデータを同期するために、先ほど説明した補助関数 allreduce と split_and_load を使う。並列性を実現するために特別なコードを書く必要がないことに注意しよう。ミニバッチ内では計算グラフにデバイス間の依存関係がないため、計算は 自動的に並列実行される。

```
def train_batch(X, y, device_params, devices, lr):
    X_shards, y_shards = split_batch(X, y, devices)
    # Loss is calculated separately on each GPU
    ls = [loss(lenet(X_shard, device_W), y_shard).sum()
          for X_shard, y_shard, device_W in zip(
              X_shards, y_shards, device_params)]
    for l in ls:  # Backpropagation is performed separately on each GPU
        l.backward()
    # Sum all gradients from each GPU and broadcast them to all GPUs
    with torch.no_grad():
        for i in range(len(device_params[0])):
            allreduce([device_params[c][i].grad for c in range(len(devices))])
    # The model parameters are updated separately on each GPU
    for param in device_params:
        d2l.sgd(param, lr, X.shape[0]) # Here, we use a full-size batch
```

```
def train_batch(X, y, device_params, devices, lr):
    X_shards, y_shards = split_batch(X, y, devices)
    with autograd.record():  # Loss is calculated separately on each GPU
        ls = [loss(lenet(X_shard, device_W), y_shard)
              for X_shard, y_shard, device_W in zip(
                  X_shards, y_shards, device_params)]
    for l in ls:  # Backpropagation is performed separately on each GPU
        l.backward()
    # Sum all gradients from each GPU and broadcast them to all GPUs
    for i in range(len(device_params[0])):
        allreduce([device_params[c][i].grad for c in range(len(devices))])
    # The model parameters are updated separately on each GPU
    for param in device_params:
        d2l.sgd(param, lr, X.shape[0])  # Here, we use a full-size batch
```

次に、学習関数を定義できる。これは前の章で使ったものとは少し異なる。GPU を割り当て、すべてのモデルパラメータをすべてのデバイスにコピーする必要があるからである。

もちろん、各バッチは複数 GPU を扱うために train_batch 関数で処理される。便宜上（そしてコードを簡潔にするため）、精度の計算は 1 枚の GPU 上で行うが、他の GPU はアイドル状態になるため、これは 非効率的である。

```
def train(num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    devices = [d2l.try_gpu(i) for i in range(num_gpus)]
    # Copy model parameters to `num_gpus` GPUs
    device_params = [get_params(params, d) for d in devices]
    num_epochs = 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
        timer.start()
```

(continues on next page)

(continued from previous page)

```
        for X, y in train_iter:
            # Perform multi-GPU training for a single minibatch
            train_batch(X, y, device_params, devices, lr)
            torch.cuda.synchronize()
        timer.stop()
        # Evaluate the model on GPU 0
        animator.add(epoch + 1, (d2l.evaluate_accuracy_gpu(
            lambda x: lenet(x, device_params[0]), test_iter, devices[0]),))
    print(f'test acc: {animator.Y[0][-1]:.2f}, {timer.avg():.1f} sec/epoch '
          f'on {str(devices)}')
```

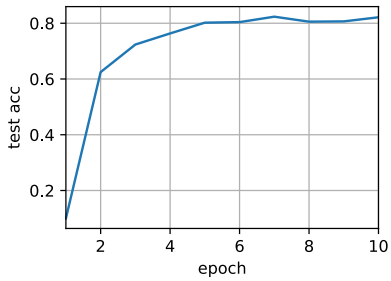
```
def train(num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    devices = [d2l.try_gpu(i) for i in range(num_gpus)]
    # Copy model parameters to `num_gpus` GPUs
    device_params = [get_params(params, d) for d in devices]
    num_epochs = 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
        timer.start()
        for X, y in train_iter:
            # Perform multi-GPU training for a single minibatch
            train_batch(X, y, device_params, devices, lr)
            npx.waitall()
        timer.stop()
        # Evaluate the model on GPU 0
        animator.add(epoch + 1, (d2l.evaluate_accuracy_gpu(
            lambda x: lenet(x, device_params[0]), test_iter, devices[0]),))
    print(f'test acc: {animator.Y[0][-1]:.2f}, {timer.avg():.1f} sec/epoch '
          f'on {str(devices)}')
```

これが単一 GPU 上でどの程度うまく動くか見てみよう。
まず、バッチサイズを 256、学習率を 0.2 にする。

```
train(num_gpus=1, batch_size=256, lr=0.2)
```

```
test acc: 0.82, 3.7 sec/epoch on [device(type='cuda', index=0)]
```

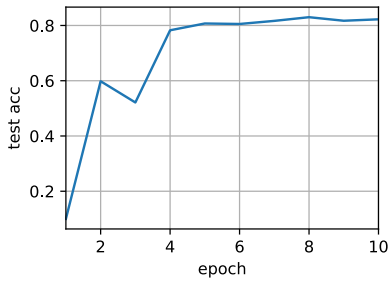
バッチサイズと学習率を変えずに GPU 数を 2 に増やすと、テスト精度は前の実験と比べておおむね同じままであることがわかる。
最適化アルゴリズムの観点では、両者は同一である。残念ながら、ここでは意味のある高速化は得られない。モデルが単純に小さすぎるうえ、データセットも小さいためである。また、複数 GPU 学習の実装にやや洗練されていない方法



を使っているため、Python のオーバーヘッドも大きくなっている。今後は、より複雑なモデルと、より洗練された並列化の方法に出会うことになる。
とはいえ、Fashion-MNIST で何が起こるか見てみよう。

```
train(num_gpus=2, batch_size=256, lr=0.2)
```

```
test acc: 0.82, 3.8 sec/epoch on [device(type='cuda', index=0), device(type=
↪'cuda', index=1)]
```



16.5.7 まとめ

- 深層ネットワークの学習を複数 GPU に分割する方法はいくつかある。層間で分割する方法、層ごとに分割する方法、データに分割する方法である。前の 2 つは、綿密に調整されたデータ転送を必要とする。データ並列が最も単純な戦略である。
- データ並列学習は簡単である。ただし、効率を高めるために実効ミニバッチサイズを増やする。
- データ並列では、データを複数 GPU に分割し、各 GPU が独自に順伝播と逆伝播を実行し、その後で勾配を集約して結果を GPU にブロードキャストする。
- より大きなミニバッチに対しては、学習率を少し大きくしてもよいだろう。

16.5.8 演習

1. k 枚の GPU で学習するとき、ミニバッチサイズを b から $k \cdot b$ に変更し、つまり GPU 数に応じて拡大しなさい。
2. 異なる学習率で精度を比較しなさい。GPU 数に応じてどのようにスケールするか。
3. 異なる GPU 上の異なるパラメータを集約する、より効率的な `allreduce` 関数を実装しなさい。なぜそれがより効率的なのか。
4. 複数 GPU でのテスト精度計算を実装しなさい。

16.6 複数 GPU のための簡潔な実装

新しいモデルごとに並列化を一から実装するのは楽しいものではない。さらに、高性能を実現するためには同期ツールの最適化に大きな利点がある。以下では、深層学習フレームワークの高レベル API を使ってこれを行う方法を示す。数学とアルゴリズムは Section 16.5 と同じである。当然ながら、この節のコードを実行するには少なくとも 2 つの GPU が必要である。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

16.6.1 おもちゃのネットワーク

Section 16.5 の LeNet よりも少し意味のある、それでいて十分に簡単かつ高速に学習できるネットワークを使いましょう。ここでは ResNet-18 の変種 (He *et al.*, 2016) を選ぶ。入力画像が小さいので、少し変更を加える。特に、Section 11.6 との違いは、最初の畳み込みでより小さいカーネル、ストライド、パディングを使うことである。さらに、最大プーリング層を取り除きる。

```
#@save
def resnet18(num_classes, in_channels=1):
    """A slightly modified ResNet-18 model."""
    def resnet_block(in_channels, out_channels, num_residuals,
                     first_block=False):
        blk = []
```

(continues on next page)

(continued from previous page)

```
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.append(d2l.Residual(out_channels, use_1x1conv=True,
                                        strides=2))
            else:
                blk.append(d2l.Residual(out_channels))
        return nn.Sequential(*blk)

    # This model uses a smaller convolution kernel, stride, and padding and
    # removes the max-pooling layer
    net = nn.Sequential(
        nn.Conv2d(in_channels, 64, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU())
    net.add_module("resnet_block1", resnet_block(64, 64, 2, first_block=True))
    net.add_module("resnet_block2", resnet_block(64, 128, 2))
    net.add_module("resnet_block3", resnet_block(128, 256, 2))
    net.add_module("resnet_block4", resnet_block(256, 512, 2))
    net.add_module("global_avg_pool", nn.AdaptiveAvgPool2d((1,1)))
    net.add_module("fc", nn.Sequential(nn.Flatten(),
                                       nn.Linear(512, num_classes)))
    return net
```

```
#@save
def resnet18(num_classes):
    """A slightly modified ResNet-18 model."""
    def resnet_block(num_channels, num_residuals, first_block=False):
        blk = nn.Sequential()
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.add(d2l.Residual(
                    num_channels, use_1x1conv=True, strides=2))
            else:
                blk.add(d2l.Residual(num_channels))
        return blk

    net = nn.Sequential()
    # This model uses a smaller convolution kernel, stride, and padding and
    # removes the max-pooling layer
    net.add(nn.Conv2D(64, kernel_size=3, strides=1, padding=1),
            nn.BatchNorm(), nn.Activation('relu'))
    net.add(resnet_block(64, 2, first_block=True),
            resnet_block(128, 2),
            resnet_block(256, 2),
            resnet_block(512, 2))
    net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
    return net
```

16.6.2 ネットワークの初期化

学習ループの中でネットワークを初期化する。初期化手法の復習については Section 8.4 を参照しよ。

```
net = resnet18(10)
# Get a list of GPUs
devices = d2l.try_all_gpus()
# We will initialize the network inside the training loop
```

```
net = resnet18(10)
# Get a list of GPUs
devices = d2l.try_all_gpus()
# Initialize all the parameters of the network
net.initialize(init=init.Normal(sigma=0.01), ctx=devices)
```

```
[07:09:59] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
[07:09:59] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
[07:10:00] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

16.6.3 学習

これまでと同様に、効率的な並列化のためには学習コードがいくつかの基本機能を実行する必要がある。

- ネットワークのパラメータをすべてのデバイス上で初期化する必要がある。
- データセットを反復する間、ミニバッチをすべてのデバイスに分割する必要がある。
- 損失とその勾配をデバイス間で並列に計算する。
- 勾配を集約し、それに応じてパラメータを更新する。

最後に、ネットワークの最終性能を報告するために、精度を（再び並列に）計算する。学習ルーチンは、データの分割と集約が必要である点を除けば、前の章の実装とかなり似ている。

```
def train(net, num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    devices = [d2l.try_gpu(i) for i in range(num_gpus)]
    def init_weights(module):
        if type(module) in [nn.Linear, nn.Conv2d]:
            nn.init.normal_(module.weight, std=0.01)
    net.apply(init_weights)
    # Set the model on multiple GPUs
    net = nn.DataParallel(net, device_ids=devices)
    trainer = torch.optim.SGD(net.parameters(), lr)
    loss = nn.CrossEntropyLoss()
    timer, num_epochs = d2l.Timer(), 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    for epoch in range(num_epochs):
```

(continues on next page)

(continued from previous page)

```
        net.train()
        timer.start()
        for X, y in train_iter:
            trainer.zero_grad()
            X, y = X.to(devices[0]), y.to(devices[0])
            l = loss(net(X), y)
            l.backward()
            trainer.step()
        timer.stop()
        animator.add(epoch + 1, (d2l.evaluate_accuracy_gpu(net, test_iter),))
    print(f'test acc: {animator.Y[0][-1]:.2f}, {timer.avg():.1f} sec/epoch '
          f'on {str(devices)}')
```

```
x = np.random.uniform(size=(4, 1, 28, 28))
x_shards = gluon.utils.split_and_load(x, devices)
net(x_shards[0]), net(x_shards[1])
```

```
[07:10:01] ../src/operator/cudnn_ops.cc:318: Auto-tuning cuDNN op, set MXNET_
↪CUDNN_AUTOTUNE_DEFAULT to 0 to disable
[07:10:02] ../src/operator/cudnn_ops.cc:318: Auto-tuning cuDNN op, set MXNET_
↪CUDNN_AUTOTUNE_DEFAULT to 0 to disable
[07:10:03] ../src/operator/cudnn_ops.cc:318: Auto-tuning cuDNN op, set MXNET_
↪CUDNN_AUTOTUNE_DEFAULT to 0 to disable
[07:10:04] ../src/operator/cudnn_ops.cc:318: Auto-tuning cuDNN op, set MXNET_
↪CUDNN_AUTOTUNE_DEFAULT to 0 to disable
```

```
(array([[ 2.2610207e-06,  2.2045981e-06, -5.4046786e-06,  1.2869955e-06,
          5.1373163e-06, -3.8297967e-06,  1.4339059e-07,  5.4683451e-06,
         -2.8279192e-06, -3.9651104e-06],
        [ 2.0698672e-06,  2.0084667e-06, -5.6382510e-06,  1.0498458e-06,
          5.5506434e-06, -4.1065491e-06,  6.0830087e-07,  5.4521784e-06,
         -3.7365021e-06, -4.1891640e-06]], ctx=gpu(0)),
 array([[ 2.4629783e-06,  2.6015525e-06, -5.4362617e-06,  1.2938218e-06,
          5.6387889e-06, -4.1360108e-06,  3.5758853e-07,  5.5125256e-06,
         -3.1957325e-06, -4.2976326e-06],
        [ 1.9431673e-06,  2.2600434e-06, -5.2698201e-06,  1.4807417e-06,
          5.4830934e-06, -3.9678889e-06,  7.5751018e-08,  5.6764356e-06,
         -3.2530229e-06, -4.0943951e-06]], ctx=gpu(1)))
```

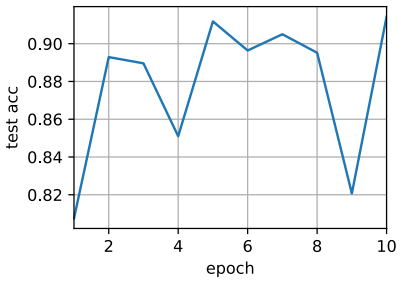
実際にどのように動作するか見てみよう。準備として、ネットワークを単一GPUで学習する。

```
train(net, num_gpus=1, batch_size=256, lr=0.1)
```

```
test acc: 0.91, 12.3 sec/epoch on [device(type='cuda', index=0)]
```

```
weight = net[0].params.get('weight')
```

(continues on next page)



(continued from previous page)

```
try:
    weight.data()
except RuntimeError:
    print('not initialized on cpu')
weight.data(devices[0])[0], weight.data(devices[1])[0]
```

```
not initialized on cpu
```

```
(array([[[ 0.01382882, -0.01183044,  0.01417865],
         [-0.00319718,  0.00439528,  0.02562625],
         [-0.00835081,  0.01387452, -0.01035946]]], ctx=gpu(0)),
 array([[[ 0.01382882, -0.01183044,  0.01417865],
         [-0.00319718,  0.00439528,  0.02562625],
         [-0.00835081,  0.01387452, -0.01035946]]], ctx=gpu(1)))
```

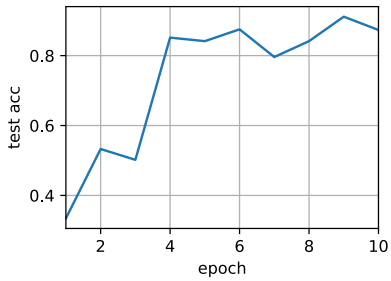
次に、学習に 2 つの GPU を使う。Section 16.5 で評価した LeNet と比べると、ResNet-18 のモデルはかなり複雑である。ここで並列化の利点が現れる。計算時間はパラメータ同期の時間よりも十分に大きくなる。これにより、並列化のオーバーヘッドの相対的な影響が小さくなるため、スケーラビリティが向上する。

```
train(net, num_gpus=2, batch_size=512, lr=0.2)
```

```
test acc: 0.87, 7.5 sec/epoch on [device(type='cuda', index=0), device(type=
↪'cuda', index=1)]
```

```
#@save
def evaluate_accuracy_gpus(net, data_iter, split_f=d2l.split_batch):
    """Compute the accuracy for a model on a dataset using multiple GPUs."""
    # Query the list of devices
    devices = list(net.collect_params().values())[0].list_ctx()
    # No. of correct predictions, no. of predictions
    metric = d2l.Accumulator(2)
    for features, labels in data_iter:
```

(continues on next page)



(continued from previous page)

```
        X_shards, y_shards = split_f(features, labels, devices)
        # Run in parallel
        pred_shards = [net(X_shard) for X_shard in X_shards]
        metric.add(sum(float(d2l.accuracy(pred_shard, y_shard)) for
                       pred_shard, y_shard in zip(
                           pred_shards, y_shards)), labels.size)
    return metric[0] / metric[1]
```

16.6.4 まとめ

- データは、データが存在するデバイス上で自動的に評価される。
- そのデバイス上のパラメータにアクセスしようとする前に、各デバイス上でネットワークを初期化するよう注意しよう。そうしないとエラーが発生する。
- 最適化アルゴリズムは、複数 GPU にわたって自動的に集約する。

16.6.5 演習

1. この節では ResNet-18 を使った。エポック数、バッチサイズ、学習率を変えてみよ。計算により多くの GPU を使ってみよう。16 個の GPU（たとえば AWS p2.16xlarge インスタンス）で試すとどうなるか？
2. ときには、デバイスごとに計算能力が異なることがある。GPU と CPU を同時に使うこともできる。どのように仕事を分担すべきだろうか。その労力に見合う価値はあるだろうか。なぜですか？なぜではないですか？

16.7 パラメータサーバー

単一 GPU から複数 GPU へ、さらに複数 GPU を含む複数サーバーへと移行し、それらが複数のラックやネットワークスイッチにまたがって配置される可能性があるにつれて、分散・並列学習のためのアルゴリズムは、より高度なものに

する必要がある。詳細は重要である。というのも、相互接続方式によって帯域幅は大きく異なるからである（たとえば、NVLink は適切な設定では 6 本のリンクを通じて最大 100 GB/s を提供でき、PCIe 4.0（16 レーン）は 32 GB/s を提供する。一方、高速な 100GbE Ethernet でさえ 10 GB/s にすぎない）。同時に、統計モデリングの専門家にネットワークやシステムの専門家であることを期待するのは無理がある。

パラメータサーバーの中核となるアイデアは、分散潜在変数モデルの文脈で Smola and Narayanamurthy (2010) により導入された。その後、push と pull の意味論について Ahmed *et al.* (2012) で説明され、システムとオープンソースライブラリについて Li *et al.* (2014) で説明された。以下では、効率化に必要な構成要素を動機づける。

16.7.1 データ並列学習

分散学習におけるデータ並列学習のアプローチを見直しよう。この節では、他のすべての方法を除外してこれを用いる。というのも、実際の実装がかなり単純だからである。GPU には現在十分なメモリがあるため、（グラフ上の深層学習を除けば）他の並列化戦略が好まれる実用例はほとんどない。Fig. 16.7.1 は、Section 16.5 で実装したデータ並列性の変種を示している。その要点は、更新されたパラメータをすべての GPU に再配布する前に、勾配の集約を 1 枚の GPU（GPU 0）上で行うことである。

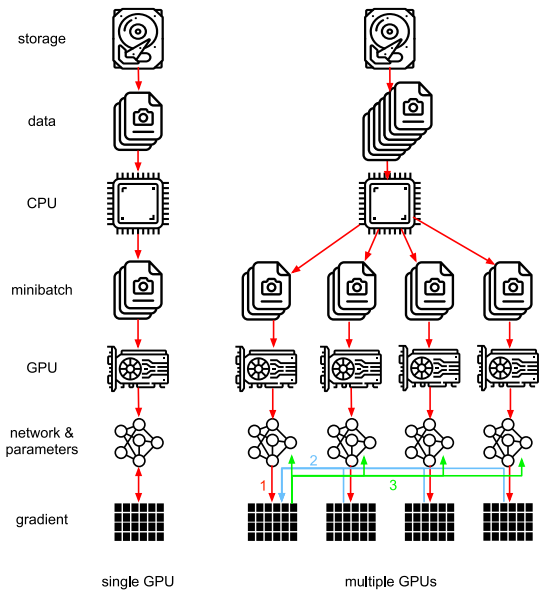


Fig. 16.7.1 左: 単一 GPU での学習。右: マルチ GPU 学習の変種。(1) 損失と勾配を計算し、(2) すべての勾配を 1 枚の GPU 上で集約し、(3) パラメータ更新を行い、パラメータをすべての GPU に再配布する。

振り返ってみると、GPU 0 上で集約するという判断はかなり場当たり的に見え

る。結局のところ、CPU 上で集約してもよいはずである。実際、あるパラメータは 1 枚の GPU 上で、別のパラメータは別の GPU 上で集約することさえできる。最適化アルゴリズムがそれをサポートしている限り、そうできない理由は特にない。たとえば、勾配 $\mathbf{g}_1, \ldots, \mathbf{g}_4$ に対応する 4 つのパラメータベクトルがあるなら、それぞれの $\mathbf{g}_i$（$i = 1, \ldots, 4$）について、勾配を 1 枚の GPU 上で集約できる。

この考え方は恣意的で軽率に見えるかもしれない。確かに、数学的にはどこでも同じである。しかし、Section 16.4 で述べたように、私たちは帯域幅の異なるバスを持つ現実の物理ハードウェアを扱っている。Fig. 16.7.2 に示すような実際の 4-way GPU サーバーを考えてみよう。特に接続が良ければ、100 GbE のネットワークカードを備えているかもしれない。より一般的なのは 1–10 GbE の範囲で、有効帯域幅は 100 MB/s から 1 GB/s 程度です。CPU にはすべての GPU に直接接続するのに十分な PCIe レーンがないため（たとえば、コンシューマ向け Intel CPU では 24 レーンです）、マルチプレクサ[110] が必要になる。CPU から 16x Gen3 リンクでの帯域幅は 16 GB/s である。これは、各 GPU がスイッチに接続される速度でもある。つまり、デバイス間で通信するほうがより効率的なのである。

110



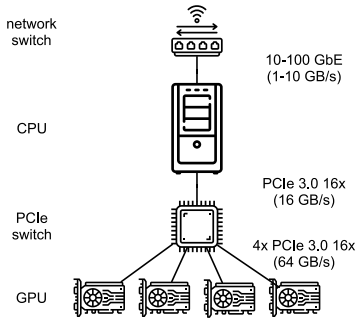


Fig. 16.7.2 4-way GPU サーバー。

議論のために、勾配のサイズが 160 MB だと仮定しよう。この場合、残りの 3 枚の GPU から 4 枚目の GPU に勾配を送るのに 30 ms かかります（各転送は 10 ms = 160 MB / 16 GB/s）。さらに重みベクトルを送り返すのに 30 ms かかるので、合計 60 ms になる。すべてのデータを CPU に送ると、4 枚の GPU それぞれが CPU にデータを送る必要があるため 40 ms の追加コストが発生し、合計 80 ms になる。最後に、勾配を 40 MB ずつ 4 つに分割できると仮定しよう。すると、PCIe スイッチはすべてのリンク間でフル帯域幅の通信を提供するので、各部分を別々の GPU 上で 同時に集約できる。30 ms の代わりに 7.5 ms で済み、同期操作全体は 15 ms になる。要するに、パラメータをどのように同期するかによって、同じ操作が 15 ms から 80 ms まで変わり得るのである。Fig. 16.7.3 は、パラメータ交換のさまざまな戦略を示している。

性能向上のために利用できる別の手段があることにも注意しよう。深いネットワークでは、上から下までのすべての勾配を計算するのにある程度の時間がかかりる。したがって、他のパラメータ群の勾配を計算している最中であって

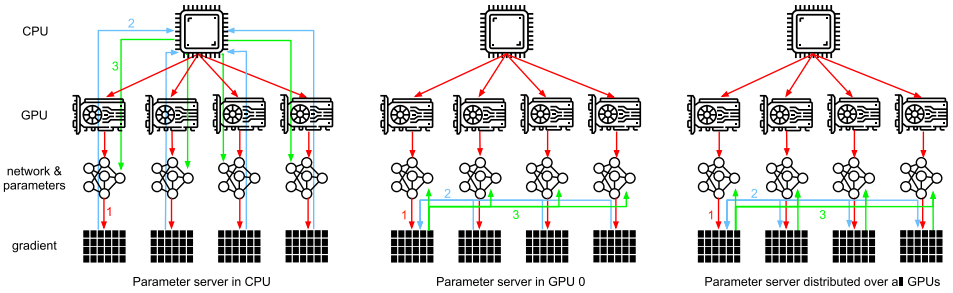


Fig. 16.7.3 パラメータ同期戦略。

も、いくつかのパラメータ群については同期を開始できる。これを Horovod でどのように行うかについては、たとえば Sergeev and Del Balso (2018) を参照しよ。

16.7.2 リング同期

現代の深層学習ハードウェアで同期を行う際には、しばしばかなり特殊なネットワーク接続に遭遇する。たとえば、AWS p3.16xlarge と NVIDIA DGX-2 のインスタンスは Fig. 16.7.4 の接続構造を共有している。各 GPU は PCIe リンクを介してホスト CPU に接続されており、その速度は最大でも 16 GB/s である。さらに各 GPU には 6 本の NVLink 接続があり、それぞれが双方向で 300 Gbit/s の転送能力を持つ。これは 1 リンク・1 方向あたり約 18 GB/s に相当する。要するに、NVLink の総帯域幅は PCIe の帯域幅よりかなり大きいのである。問題は、それを最も効率的に使う方法である。

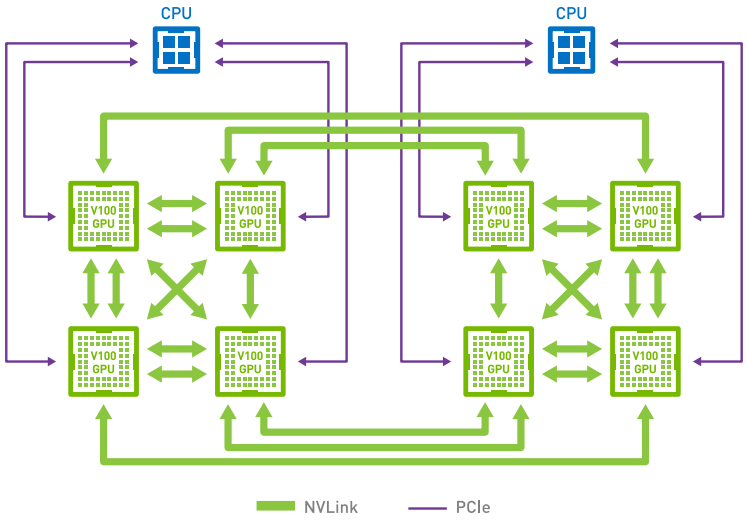


Fig. 16.7.4 8 枚の V100 GPU サーバーにおける NVLink 接続（画像提供: NVIDIA）。

最適な同期戦略は、ネットワークを 2 つのリングに分解し、それらを使ってデータを直接同期することだと分かっている (Wang *et al.*, 2018)。Fig. 16.7.5 は、ネ

ットワークが 1 つのリング（1-2-3-4-5-6-7-8-1、NVLink 帯域幅が 2 倍）と、もう 1 つのリング（1-4-6-3-5-8-2-7-1、通常の帯域幅）に分解できることを示している。この場合に効率的な同期プロトコルを設計するのは容易ではない。

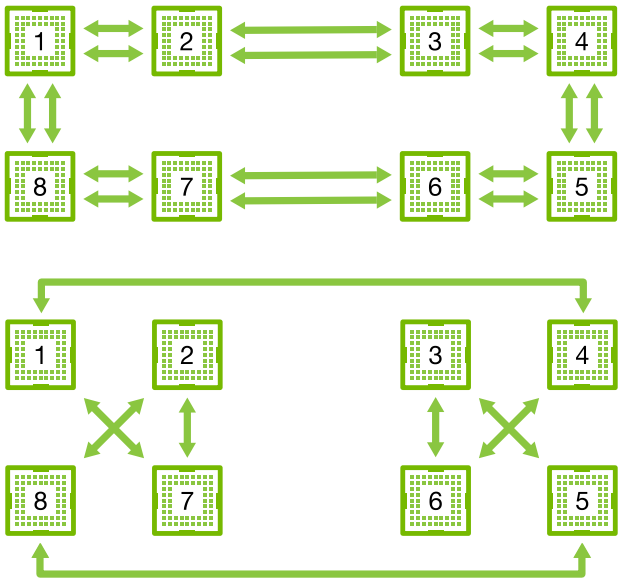


Fig. 16.7.5 NVLink ネットワークの 2 つのリングへの分解。

次の思考実験を考えてみよう。n 個の計算ノード（または GPU）からなるリングがあるとする。最初のノードから 2 番目のノードへ勾配を送ることができる。そこでそれはローカル勾配に加えられ、3 番目のノードへ送られ、以下同様に続く。$n-1$ ステップ後には、集約された勾配は最後に訪れたノードに存在する。つまり、勾配の集約にかかる時間はノード数に対して線形に増加する。しかし、この方法ではアルゴリズムはかなり非効率である。結局のところ、どの時点でも通信しているノードは 1 つしかない。もし勾配を n 個のチャンクに分割し、チャンク i の同期をノード i から開始したらどうだろうか。各チャンクのサイズは $1/n$ なので、総時間は $(n-1)/n \approx 1$ になる。言い換えると、勾配の集約に費やす時間は、リングのサイズを大きくしても 増加しないのである。これは非常に驚くべき結果である。Fig. 16.7.6 は、$n = 4$ ノードでの手順を示している。

8 枚の V100 GPU 間で 160 MB を同期する同じ例を用いると、およそ $2 \cdot 160\text{MB}/(3 \cdot 18\text{GB/s}) \approx 6\text{ms}$ になる。これは、8 枚の GPU を使っているにもかかわらず PCIe バスを使うよりも優れている。実際には、深層学習フレームワークが通信を大きなバースト転送にまとめることに失敗することが多いため、これらの数値は少し悪くなる。

リング同期が他の同期アルゴリズムと本質的に異なるという誤解がよくあるが、注意しよう。違いは、単純な木構造と比べて同期経路がやや複雑であるという点だけである。

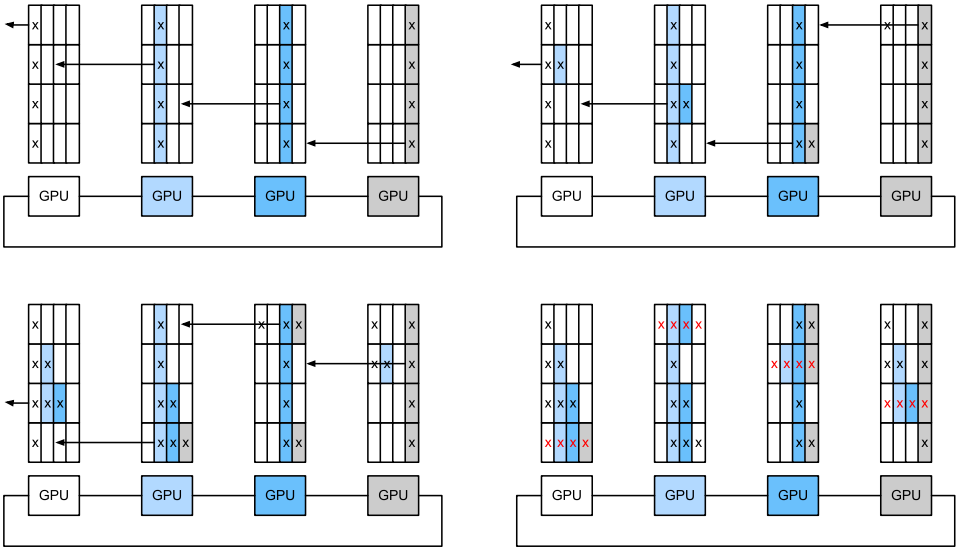


Fig. 16.7.6 4 ノード間のリング同期。各ノードは、組み上がった勾配が右隣のノードに現れるまで、勾配の一部を左隣へ送り始める。

16.7.3 マルチマシン学習

複数マシンでの分散学習には、さらに別の課題がある。すなわち、場合によっては 1 桁以上遅い、比較的低帯域幅のファブリックを介して接続されたサーバーと通信する必要があるのである。デバイス間の同期は厄介である。結局のところ、学習コードを実行している異なるマシンは、微妙に異なる速度で動作する。したがって、同期的な分散最適化を使いたいなら、それらを 同期させる必要がある。Fig. 16.7.7 は、分散並列学習がどのように行われるかを示している。

1. 各マシンで（異なる）バッチのデータを読み込み、それを複数の GPU に分割して GPU メモリへ転送する。そこで各 GPU バッチごとに予測と勾配を別々に計算する。
2. すべてのローカル GPU からの勾配を 1 枚の GPU 上で集約する（あるいは、その一部を異なる GPU 上で集約する）。
3. 勾配を CPU に送る。
4. CPU は勾配を中央のパラメータサーバーに送り、そこで全勾配を集約する。
5. 集約された勾配を用いてパラメータを更新し、更新後のパラメータを各 CPU にブロードキャストし返す。
6. その情報を 1 枚（または複数枚）の GPU に送る。
7. 更新されたパラメータをすべての GPU に展開する。

これらの操作はどれもかなり単純に見える。実際、単一マシン 内では効率的に

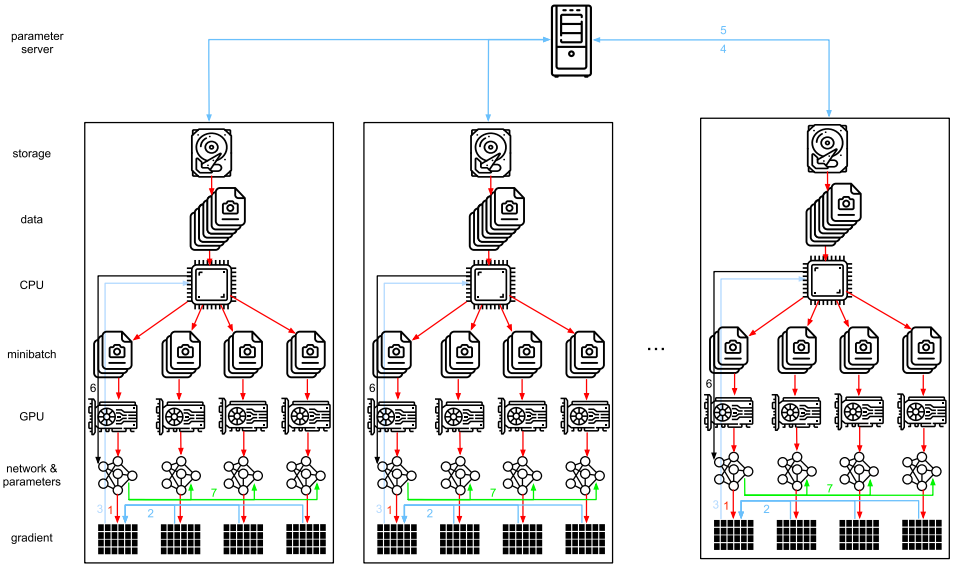


Fig. 16.7.7 マルチマシン・マルチ GPU 分散並列学習。

実行できる。しかし複数マシンになると、中央のパラメータサーバーがボトルネックになることが分かりる。結局のところ、サーバーごとの帯域幅には限りがあるため、m 個のワーカーに対してすべての勾配をサーバーへ送るのにかかる時間は $O(m)$ である。この壁を破るには、サーバー数を n に増やせばよいだろう。このとき各サーバーはパラメータの $O(1/n)$ しか保持する必要がないため、更新と最適化にかかる総時間は $O(m/n)$ になる。両者を対応させれば、扱うワーカー数に関係なく一定のスケーリングが得られる。実際には、同じマシンをワーカーとしてもサーバーとしても使う。Fig. 16.7.8 はその設計を示している（詳細は (Li *et al.*, 2014) も参照されたい）。特に、複数マシンが不合理な遅延なく動作するようにするのは容易ではない。

16.7.4 キー–バリューストア

分散マルチ GPU 学習に必要な手順を実際に実装するのは容易ではない。そのため、共通の抽象化、すなわち更新の意味論を再定義した キー–バリューストアを使うと有益である。

多くのワーカーと多くの GPU にまたがって、勾配 i の計算は次のように定義できる。

$$\mathbf{g}_i = \sum_{k \in \text{workers}} \sum_{j \in \text{GPUs}} \mathbf{g}_{ijk}, \tag{16.7.1}$$

ここで $\mathbf{g}_{ijk}$ は、ワーカー k の GPU j 上で分割された勾配 i の一部である。この操作の要点は、これが 可換な縮約であることである。つまり、多くのベクトルを 1 つにまとめ、どの順序で操作を適用しても結果が変わらない。これは私たちの目的にとって都合がよい性質である。というのも、どの勾配がいつ受信さ

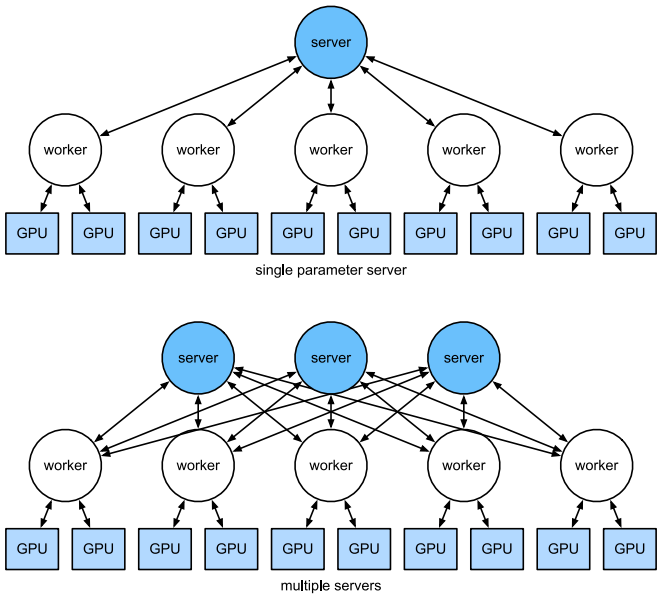


Fig. 16.7.8 上: 単一のパラメータサーバーは帯域幅が有限なのでボトルネックになる。下: 複数のパラメータサーバーが、総帯域幅を持つパラメータの一部を保持する。

れるかを細かく制御する必要が（なくても）よいからである。さらに、この操作は異なる i の間で独立であることにも注意しよう。

これにより、次の 2 つの操作を定義できる。勾配を蓄積する *push* と、集約された勾配を取得する *pull* である。勾配の集合は多数存在するため（結局のところ、層も多数ある）、勾配をキー i で索引付けする必要がある。Dynamo (DeCandia *et al.*, 2007) で導入されたもののようなキー–バリューストアとのこの類似性は偶然ではない。キー–バリューストアもまた、多くの類似した特性を満たしており、特に複数サーバーにパラメータを分散する際にそうです。

キー–バリューストアにおける push と pull の操作は次のように説明される。

- **push(key, value)** は、特定の勾配（value）をワーカーから共通ストレージへ送りる。そこでその値は、たとえば加算によって集約される。
- **pull(key, value)** は、共通ストレージから集約済みの値を取得する。たとえば、すべてのワーカーからの勾配をまとめた後に取得する。

同期に関する複雑さをすべて単純な push と pull の操作の背後に隠すことで、最適化を単純な形で表現したい統計モデラーの関心事と、分散同期に本質的な複雑さを扱わなければならないシステムエンジニアの関心事を分離できる。

16.7.5 まとめ

- 同期は、特定のネットワーク基盤やサーバー内の接続性に対して高度に適応的である必要がある。これにより、同期にかかる時間が大きく変わり得る。
- リング同期は p3 および DGX-2 サーバーでは最適になり得るが、他ではそうとは限らない。
- 複数のパラメータサーバーを追加して帯域幅を増やす場合には、階層的な同期戦略がうまく機能する。

16.7.6 演習

1. リング同期をさらに高速化できるか？ヒント: 両方向にメッセージを送ることができる。
2. 非同期通信を許可することは可能ですか（計算がまだ進行中の間に）？それは性能にどのように影響するか？
3. 長時間実行される計算の途中でサーバーを失ったらどうなるか？計算を完全にやり直さずに済むような フォールトトレランス機構をどのように設計できるか？

17 コンピュータビジョン

医療診断、自動運転車、カメラ監視、スマートフィルターなど、コンピュータビジョン分野の多くの応用は、私たちの現在および将来の生活と密接に関係している。

近年、深層学習はコンピュータビジョンシステムの性能向上をもたらす変革的な力となってきた。

最先端のコンピュータビジョン応用のほとんどは、深層学習と切り離せないと言ってよいだろう。

このことを踏まえ、本章ではコンピュータビジョン分野に焦点を当て、学術界および産業界で近年影響力のあった手法と応用を調べる。

Section 10 と Section 11 では、コンピュータビジョンで一般的に用いられるさまざまな畳み込みニューラルネットワークを学び、それらを単純な画像分類タスクに適用した。

本章の冒頭では、モデルの汎化性能を改善する可能性のある 2 つの手法、すなわち 画像拡張と ファインチューニングを説明し、それらを画像分類に適用する。

深層ニューラルネットワークは画像を複数のレベルで効果的に表現できるため、このような層ごとの表現は 物体検出、意味的セグメンテーション、スタイル変換などのさまざまなコンピュータビジョンタスクで成功裏に利用されてきた。

コンピュータビジョンにおける層ごとの表現を活用するという重要な考え方に従い、まず物体検出の主要な構成要素と技法から始める。次に、画像の意味的セグメンテーションに 全畳み込みネットワークをどのように用いるかを示す。その後、この本の表紙のような画像を生成するためにスタイル変換技法をどのように使うかを説明する。最後に、本章の内容とこれまでのいくつかの章の材料を、2 つの人気のあるコンピュータビジョンベンチマークデータセットに適用することで、本章を締めくくりる。

17.1 画像拡張

Section 11.1 では、さまざまな応用において深層ニューラルネットワークが成功するための前提条件として、大規模データセットが必要であることを述べた。
画像拡張は、訓練画像に一連のランダムな変換を施すことで、元の画像に似ているが異なる訓練例を生成し、訓練データセットの規模を拡大する。
別の見方をすると、画像拡張は、訓練例に対するランダムな微調整によって、モデルが特定の属性に過度に依存しにくくなり、その結果として汎化能力が向上する、という事実に動機づけられている。
たとえば、画像をさまざまな方法で切り抜くことで、注目対象が異なる位置に現れるようにでき、これにより対象の位置へのモデルの依存を減らせる。
また、明るさや色などの要素を調整して、色に対するモデルの感度を下げることもできる。
当時の AlexNet の成功には、画像拡張が不可欠だった可能性が高いだろう。
この節では、コンピュータビジョンで広く使われているこの手法について説明する。

```
%matplotlib inline
from d2l import torch as d2l
import torch
import torchvision
from torch import nn
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()
```

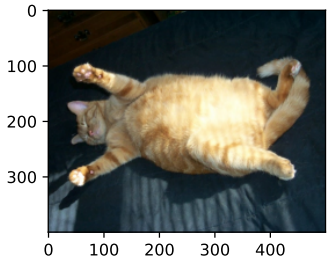
```
def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows * num_cols)]
    d2l.show_images(Y, num_rows, num_cols, scale=scale)
```

```
def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows * num_cols)]
    d2l.show_images(Y, num_rows, num_cols, scale=scale)
```

17.1.1 一般的な画像拡張手法

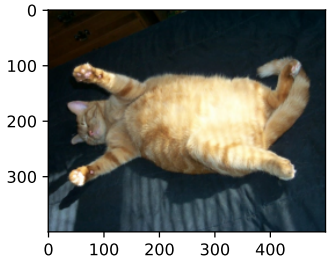
一般的な画像拡張手法を調べるにあたり、以下の 400×500 の画像を例として用いる。

```
d2l.set_figsize()
img = d2l.Image.open('../img/cat1.jpg')
d2l.plt.imshow(img);
```




```
d2l.set_figsize()
img = image.imread('../img/cat1.jpg')
d2l.plt.imshow(img.asnumpy());
```

```
[07:00:45] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```




```
train_with_data_aug(train_augs, test_augs, net)
```

```
train_with_data_aug(train_augs, test_augs, net)
```

多くの画像拡張手法には、ある程度のランダム性がある。画像拡張の効果を観察しやすくするために、まず補助関数 apply を定義する。この関数は、画像拡張手法 aug を入力画像 img に対して複数回実行し、その結果をすべて表示す。

```
def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows * num_cols)]
    d2l.show_images(Y, num_rows, num_cols, scale=scale)
```

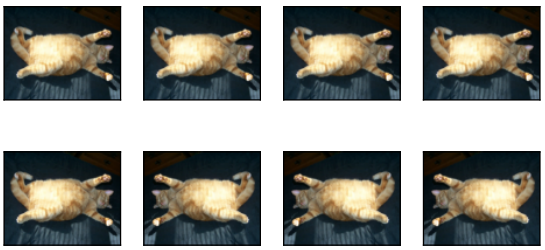
反転と切り抜き

画像を左右反転することは、通常、対象のカテゴリを変えない。
これは、最も早くから使われ、かつ最も広く用いられている画像拡張手法の一つである。
次に、transforms モジュールを使って RandomHorizontalFlip のインスタンスを作成する。これは、50% の確率で画像を左右反転する。

```
apply(img, torchvision.transforms.RandomHorizontalFlip())
```



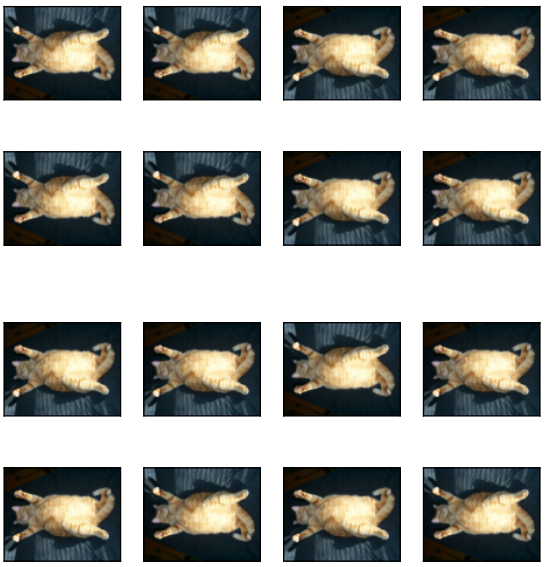
```
apply(img, gluon.data.vision.transforms.RandomFlipLeftRight())
```



上下反転することは、左右反転ほど一般的ではない。とはいえ、この例の画像では、上下反転しても認識の妨げにはならない。
次に、RandomVerticalFlip のインスタンスを作成し、50% の確率で画像を上下反転する。

```
apply(img, torchvision.transforms.RandomVerticalFlip())
```

```
apply(img, gluon.data.vision.transforms.RandomFlipTopBottom())
```



ここで用いた例の画像では猫が画像の中央にいるが、一般にはそうとは限らない。

Section 10.5 では、プーリング層によって畳み込み層の対象位置に対する感度を下げられることを説明した。

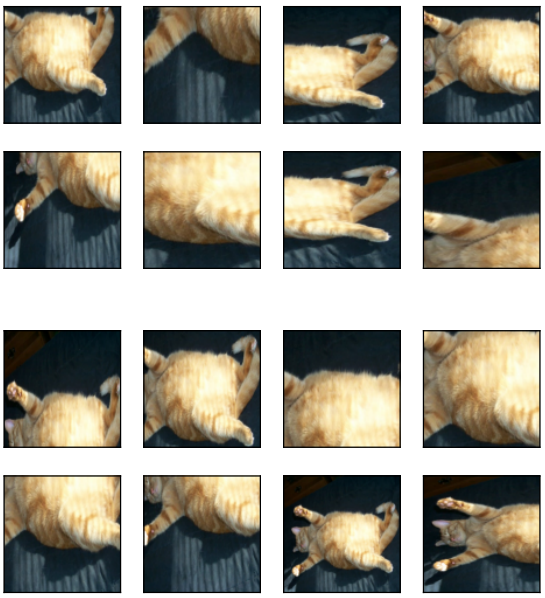
さらに、画像をランダムに切り抜くことで、対象が画像内の異なる位置や異なるスケールで現れるようにでき、これもモデルの対象位置への感度を下げるのに役立ちる。

以下のコードでは、毎回元の面積の 10% ~ 100% の領域をランダムに切り抜き、その領域の幅と高さの比を 0.5 ~ 2 の範囲からランダムに選ぶ。次に、その領域の幅と高さをどちらも 200 ピクセルに拡大縮小する。

特に断りがない限り、この節での a と b の間の乱数とは、区間 $[a, b]$ から一様分布に従ってランダムサンプリングされた連続値を指す。

```
shape_aug = torchvision.transforms.RandomResizedCrop(
    (200, 200), scale=(0.1, 1), ratio=(0.5, 2))
apply(img, shape_aug)
```

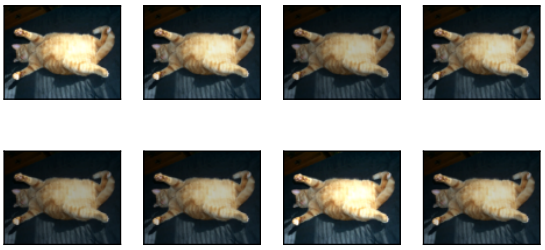
```
shape_aug = gluon.data.vision.transforms.RandomResizedCrop(
    (200, 200), scale=(0.1, 1), ratio=(0.5, 2))
apply(img, shape_aug)
```



色の変更

別の拡張手法として、色を変更する方法がある。画像の色に関して、明るさ、コントラスト、彩度、色相の 4 つの側面を変更できる。以下の例では、画像の明るさをランダムに変更し、元の画像の 50% (1 – 0.5) から 150% (1 + 0.5) の範囲の値にする。

```
apply(img, torchvision.transforms.ColorJitter(
    brightness=0.5, contrast=0, saturation=0, hue=0))
```



```
apply(img, gluon.data.vision.transforms.RandomBrightness(0.5))
```

同様に、画像の色相をランダムに変更することもできる。

```
apply(img, torchvision.transforms.ColorJitter(
    brightness=0, contrast=0, saturation=0, hue=0.5))
```



```
apply(img, gluon.data.vision.transforms.RandomHue(0.5))
```



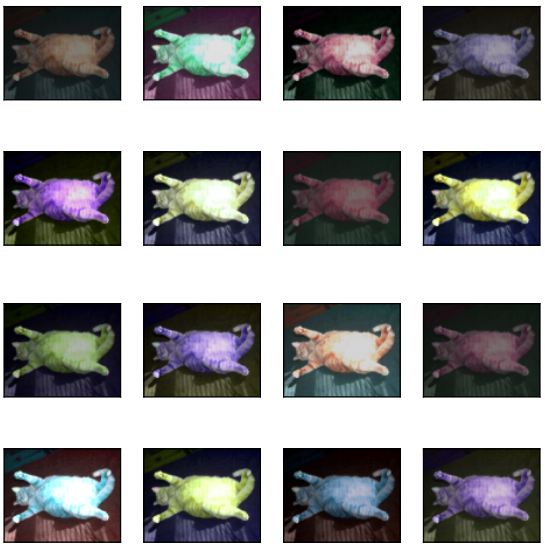
また、RandomColorJitter のインスタンスを作成し、画像の brightness、contrast、saturation、hue を同時にランダム変更する方法を設定することもできる。

```
color_aug = torchvision.transforms.ColorJitter(
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)
apply(img, color_aug)
```

```
color_aug = gluon.data.vision.transforms.RandomColorJitter(
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)
apply(img, color_aug)
```

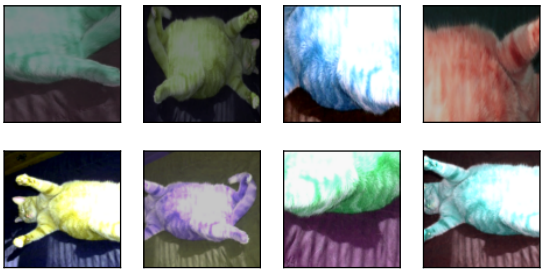
複数の画像拡張手法の組み合わせ

実際には、複数の画像拡張手法を組み合わせることがよくある。



たとえば、上で定義したさまざまな画像拡張手法を組み合わせ、Compose インスタンスを通して各画像に適用できる。

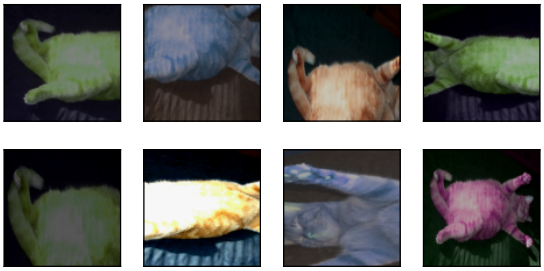
```
augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(), color_aug, shape_aug])
apply(img, augs)
```



```
augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.RandomFlipLeftRight(), color_aug, shape_aug])
apply(img, augs)
```

17.1.2 画像拡張を用いた学習

画像拡張を用いてモデルを学習してみよう。
ここでは、以前使った Fashion-MNIST データセットの代わりに CIFAR-10 データセットを使う。

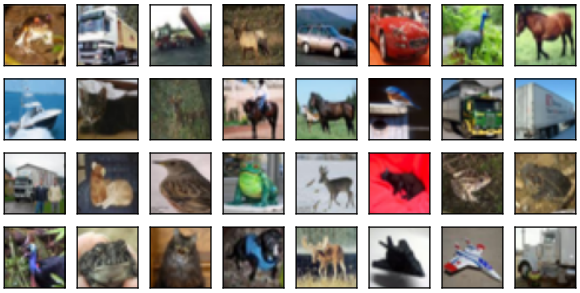


これは、Fashion-MNIST データセットでは対象の位置とサイズが正規化されている一方、CIFAR-10 データセットでは対象の色とサイズにより大きな違いがあるためである。

CIFAR-10 データセットの最初の 32 個の訓練画像を以下に示す。

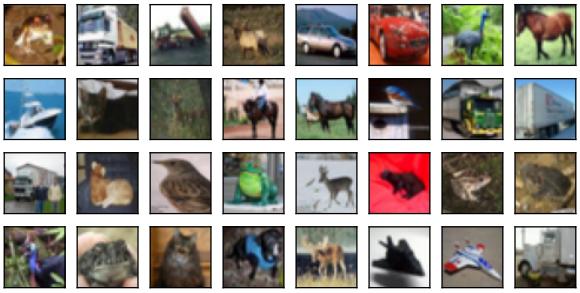
```
all_images = torchvision.datasets.CIFAR10(train=True, root="../data",
                                           download=True)
d2l.show_images([all_images[i][0] for i in range(32)], 4, 8, scale=0.8);
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ../data/
↪cifar-10-python.tar.gz
100%|          | 170498071/170498071 [02:37<00:00, 1084703.38it/s]
Extracting ../data/cifar-10-python.tar.gz to ../data
```



```
d2l.show_images(gluon.data.vision.CIFAR10(
    train=True)[:32][0], 4, 8, scale=0.8);
```

```
Downloading /opt/mxnet/datasets/cifar10/cifar-10-binary.tar.gz from https://
↪apache-mxnet.s3-accelerate.dualstack.amazonaws.com/gluon/dataset/cifar10/
↪cifar-10-binary.tar.gz...
```



予測時に確定的な結果を得るため、通常は画像拡張を訓練例にのみ適用し、予測時にはランダム操作を伴う画像拡張を使わない。

ここでは最も単純な左右反転のランダム拡張のみを使う。さらに、`ToTensor` インスタンスを使って画像のミニバッチを深層学習フレームワークが要求する形式、すなわち

形状が (バッチサイズ, チャネル数, 高さ, 幅) で、0 から 1 の間の 32 ビット浮動小数点数に変換する。

```
train_augs = torchvision.transforms.Compose([
     torchvision.transforms.RandomHorizontalFlip(),
     torchvision.transforms.ToTensor()])

test_augs = torchvision.transforms.Compose([
     torchvision.transforms.ToTensor()])
```

```
train_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    gluon.data.vision.transforms.ToTensor()])

test_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.ToTensor()])
```

次に、画像の読み込みと画像拡張の適用を容易にする補助関数を定義する。

PyTorch のデータセットが提供する `transform` 引数は、画像を変換する際に拡張を適用する。

`DataLoader` の詳細な説明については、Section 7.2 を参照しよ。

```
def load_cifar10(is_train, augs, batch_size):
    dataset = torchvision.datasets.CIFAR10(root="../data", train=is_train,
                                           transform=augs, download=True)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
```

(continues on next page)

(continued from previous page)

```
                    shuffle=is_train, num_workers=d2l.get_dataloader_workers())
    return dataloader
```

```
def load_cifar10(is_train, augs, batch_size):
    return gluon.data.DataLoader(
        gluon.data.vision.CIFAR10(train=is_train).transform_first(augs),
        batch_size=batch_size, shuffle=is_train,
        num_workers=d2l.get_dataloader_workers())
```

マルチ GPU 学習

Section 11.6 の ResNet-18 モデルを CIFAR-10 データセットで学習する。
Section 16.6 におけるマルチ GPU 学習の説明を思い出してほしい。
以下では、複数の GPU を使ってモデルを学習・評価する関数を定義する。

```
#@save
def train_batch_ch13(net, X, y, loss, trainer, devices):
    """Train for a minibatch with multiple GPUs (defined in Chapter 13)."""
    if isinstance(X, list):
        # Required for BERT fine-tuning (to be covered later)
        X = [x.to(devices[0]) for x in X]
    else:
        X = X.to(devices[0])
    y = y.to(devices[0])
    net.train()
    trainer.zero_grad()
    pred = net(X)
    l = loss(pred, y)
    l.sum().backward()
    trainer.step()
    train_loss_sum = l.sum()
    train_acc_sum = d2l.accuracy(pred, y)
    return train_loss_sum, train_acc_sum
```

```
#@save
def train_batch_ch13(net, features, labels, loss, trainer, devices,
                     split_f=d2l.split_batch):
    """Train for a minibatch with multiple GPUs (defined in Chapter 13)."""
    X_shards, y_shards = split_f(features, labels, devices)
    with autograd.record():
        pred_shards = [net(X_shard) for X_shard in X_shards]
        ls = [loss(pred_shard, y_shard) for pred_shard, y_shard
              in zip(pred_shards, y_shards)]
    for l in ls:
        l.backward()
    # The `True` flag allows parameters with stale gradients, which is useful
    # later (e.g., in fine-tuning BERT)
```

(continues on next page)

(continued from previous page)

```
    trainer.step(labels.shape[0], ignore_stale_grad=True)
    train_loss_sum = sum([float(l.sum()) for l in ls])
    train_acc_sum = sum(d2l.accuracy(pred_shard, y_shard)
                        for pred_shard, y_shard in zip(pred_shards, y_shards))
    return train_loss_sum, train_acc_sum
```

```
#@save
def train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
               devices=d2l.try_all_gpus()):
    """Train a model with multiple GPUs (defined in Chapter 13)."""
    timer, num_batches = d2l.Timer(), len(train_iter)
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 1],
                            legend=['train loss', 'train acc', 'test acc'])
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    for epoch in range(num_epochs):
        # Sum of training loss, sum of training accuracy, no. of examples,
        # no. of predictions
        metric = d2l.Accumulator(4)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = train_batch_ch13(
                net, features, labels, loss, trainer, devices)
            metric.add(l, acc, labels.shape[0], labels.numel())
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[3],
                              None))
        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {metric[0] / metric[2]:.3f}, train acc '
          f'{metric[1] / metric[3]:.3f}, test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec on '
          f'{str(devices)}')
```

```
#@save
def train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
               devices=d2l.try_all_gpus(), split_f=d2l.split_batch):
    """Train a model with multiple GPUs (defined in Chapter 13)."""
    timer, num_batches = d2l.Timer(), len(train_iter)
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 1],
                            legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        # Sum of training loss, sum of training accuracy, no. of examples,
        # no. of predictions
        metric = d2l.Accumulator(4)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = train_batch_ch13(
                net, features, labels, loss, trainer, devices, split_f)
            metric.add(l, acc, labels.shape[0], labels.size)
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
```

(continues on next page)

(continued from previous page)

```
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[3],
                              None))
        test_acc = d2l.evaluate_accuracy_gpus(net, test_iter, split_f)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {metric[0] / metric[2]:.3f}, train acc '
          f'{metric[1] / metric[3]:.3f}, test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec on '
          f'{str(devices)}')
```

ここで、画像拡張を用いてモデルを学習する `train_with_data_aug` 関数を定義する。
この関数は利用可能なすべての GPU を取得し、最適化アルゴリズムとして Adam を使い、訓練データセットに画像拡張を適用し、最後に先ほど定義した `train_ch13` 関数を呼び出してモデルを学習・評価する。

```
batch_size, devices, net = 256, d2l.try_all_gpus(), d2l.resnet18(10, 3)
net.apply(d2l.init_cnn)

def train_with_data_aug(train_augs, test_augs, net, lr=0.001):
    train_iter = load_cifar10(True, train_augs, batch_size)
    test_iter = load_cifar10(False, test_augs, batch_size)
    loss = nn.CrossEntropyLoss(reduction="none")
    trainer = torch.optim.Adam(net.parameters(), lr=lr)
    net(next(iter(train_iter))[0])
    train_ch13(net, train_iter, test_iter, loss, trainer, 10, devices)
```

```
batch_size, devices, net = 256, d2l.try_all_gpus(), d2l.resnet18(10)
net.initialize(init=init.Xavier(), ctx=devices)

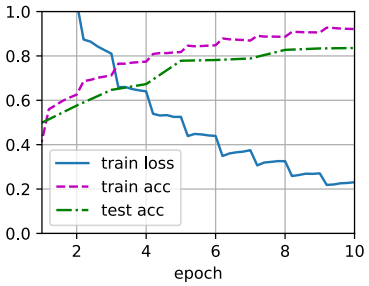
def train_with_data_aug(train_augs, test_augs, net, lr=0.001):
    train_iter = load_cifar10(True, train_augs, batch_size)
    test_iter = load_cifar10(False, test_augs, batch_size)
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    train_ch13(net, train_iter, test_iter, loss, trainer, 10, devices)
```

```
[07:00:57] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
[07:00:57] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

それでは、ランダム左右反転に基づく画像拡張を用いてモデルを学習してみよう。

```
train_with_data_aug(train_augs, test_augs, net)
```

```
loss 0.231, train acc 0.921, test acc 0.836
4857.3 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



17.1.3 まとめ

- 画像拡張は、既存の訓練データに基づいてランダムな画像を生成し、モデルの汎化能力を向上させる。
- 予測時に確定的な結果を得るため、通常は画像拡張を訓練例にのみ適用し、予測時にはランダム操作を伴う画像拡張を使わない。
- 深層学習フレームワークは、同時に適用できる多様な画像拡張手法を提供している。

17.1.4 演習

1. 画像拡張を使わずにモデルを学習せよ: `train_with_data_aug(test_augs, test_augs)`. 画像拡張を使う場合と使わない場合の訓練精度とテスト精度を比較しよ。この比較実験は、画像拡張が過学習を緩和できるという主張を支持できるか。なぜか。
2. CIFAR-10 データセットでのモデル学習に、複数の異なる画像拡張手法を組み合わせよ。テスト精度は向上するか。
3. 深層学習フレームワークのオンラインドキュメントを参照しよ。ほかにどのような画像拡張手法が提供されているか。

17.2 ファインチューニング

前の章では、Fashion-MNIST の訓練データセットに対して、わずか 60000 枚の画像だけを用いてモデルを学習する方法について説明した。また、学術分野で最も広く使われている大規模画像データセットである ImageNet についても述べた。ImageNet には 1000 種類の物体と 1000 万枚以上の画像がある。しかし、私たちが通常扱うデータセットの規模は、この 2 つのデータセットの中間にある。

画像からさまざまな種類の椅子を認識し、その後ユーザーに購入リンクを推薦したいとする。1 つの方法として、まず 100 種類の一般的な椅子を特定し、各椅子について異なる角度から 1000 枚の画像を集め、収集した画像データセット上で分類モデルを学習することが考えられる。この椅子データセットは Fashion-MNIST データセットより大きいかもしれないが、例の数は依然として ImageNet の 10 分の 1 未満である。そのため、ImageNet に適した複雑なモデルは、この椅子データセットでは過学習を起こす可能性がある。さらに、訓練例の数が限られているため、学習したモデルの精度が実用上の要求を満たさないかもしれない。

上記の問題に対処するためには、明らかな解決策として、より多くのデータを収集することが挙げられる。しかし、データの収集とラベル付けには多くの時間と費用がかかりる。たとえば、ImageNet データセットを収集するために、研究者たちは研究資金から数百万ドルを費やした。現在ではデータ収集コストは大幅に下がっているが、それでもこのコストは無視できない。

別の解決策は、転移学習を適用して、ソースデータセットで学習した知識を ターゲットデータセットに転移することである。たとえば、ImageNet データセットの画像の大半は椅子とは無関係であるが、このデータセットで学習したモデルは、より一般的な画像特徴を抽出できる可能性があり、エッジ、テクスチャ、形状、物体の構成を識別するのに役立ちる。このような類似した特徴は、椅子の認識にも有効である可能性がある。

17.2.1 手順

この節では、転移学習における一般的な手法である ファインチューニングを紹介する。Fig. 17.2.1 に示すように、ファインチューニングは次の 4 つの手順から成る。

1. ソースデータセット（たとえば ImageNet データセット）上で、ニューラルネットワークモデル、すなわち ソースモデルを事前学習する。
2. 新しいニューラルネットワークモデル、すなわち ターゲットモデルを作成する。これは出力層を除いて、ソースモデル上のすべてのモデル設計とそのパラメータをコピーする。これらのモデルパラメータにはソースデータセットで学習した知識が含まれており、その知識はターゲットデータセットにも適

用できると仮定する。また、ソースモデルの出力層はソースデータセットのラベルと密接に関係していると仮定するため、ターゲットモデルでは使用しない。

3. ターゲットモデルに出力層を追加し、その出力数をターゲットデータセットのカテゴリ数にする。そして、この層のモデルパラメータをランダムに初期化する。

4. 椅子データセットのようなターゲットデータセット上でターゲットモデルを学習する。出力層はゼロから学習され、それ以外のすべての層のパラメータはソースモデルのパラメータに基づいてファインチューニングされる。

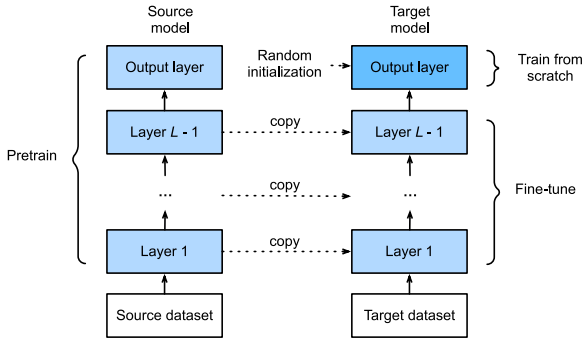


Fig. 17.2.1 Fine tuning.

ターゲットデータセットがソースデータセットよりはるかに小さい場合、ファインチューニングはモデルの汎化能力の向上に役立ちる。

17.2.2 ホットドッグ認識

具体例として、ホットドッグ認識を通じてファインチューニングを示しよう。ImageNet データセットで事前学習された ResNet モデルを、小さなデータセット上でファインチューニングする。この小さなデータセットは、ホットドッグを含む画像と含まない画像からなる数千枚の画像で構成されている。ファインチューニングしたモデルを用いて、画像からホットドッグを認識する。

```
%matplotlib inline
from d2l import torch as d2l
from torch import nn
import torch
import torchvision
import os
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn
import os
```

(continues on next page)

(continued from previous page)

```
npx.set_np()
```

```
#@save
d2l.DATA_HUB['hotdog'] = (d2l.DATA_URL + 'hotdog.zip',
                          'fba480ffa8aa7e0febbb511d181409f899b9baa5')

data_dir = d2l.download_extract('hotdog')
```


```
#@save
d2l.DATA_HUB['hotdog'] = (d2l.DATA_URL + 'hotdog.zip',
                          'fba480ffa8aa7e0febbb511d181409f899b9baa5')

data_dir = d2l.download_extract('hotdog')
```


データセットの読み込み

ここで使用するホットドッグデータセットは、オンライン画像から取得したものです。このデータセットは、ホットドッグを含む正例画像 1400 枚と、他の食品を含む同数の負例画像で構成されている。両クラスの画像のうち 1000 枚を訓練に使用し、残りをテストに使用する。

ダウンロードしたデータセットを解凍すると、hotdog/train と hotdog/test の 2 つのフォルダが得られる。どちらのフォルダにも hotdog と not-hotdog のサブフォルダがあり、それぞれ対応するクラスの画像が含まれている。


```
#@save
d2l.DATA_HUB['hotdog'] = (d2l.DATA_URL + 'hotdog.zip',
                          'fba480ffa8aa7e0febbb511d181409f899b9baa5')

data_dir = d2l.download_extract('hotdog')
```


```
Downloading ../data/hotdog.zip from http://d2l-data.s3-accelerate.amazonaws.
↪com/hotdog.zip...
```


```
#@save
d2l.DATA_HUB['hotdog'] = (d2l.DATA_URL + 'hotdog.zip',
                          'fba480ffa8aa7e0febbb511d181409f899b9baa5')

data_dir = d2l.download_extract('hotdog')
```



```
Downloading ../data/hotdog.zip from http://d2l-data.s3-accelerate.amazonaws.
↪com/hotdog.zip...
```

```
hotdogs = [train_imgs[i][0] for i in range(8)]
not_hotdogs = [train_imgs[-i - 1][0] for i in range(8)]
d2l.show_images(hotdogs + not_hotdogs, 2, 8, scale=1.4);
```

```
hotdogs = [train_imgs[i][0] for i in range(8)]
not_hotdogs = [train_imgs[-i - 1][0] for i in range(8)]
d2l.show_images(hotdogs + not_hotdogs, 2, 8, scale=1.4);
```

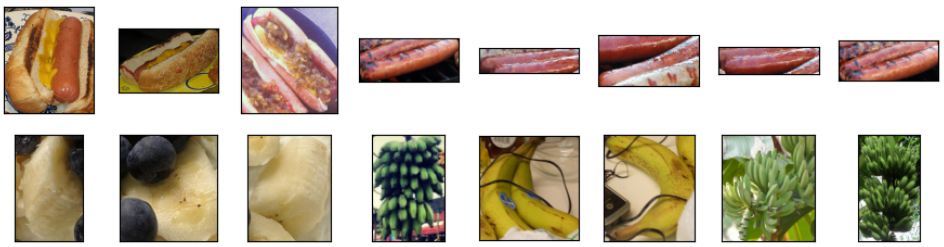
訓練データセットとテストデータセットのすべての画像ファイルをそれぞれ読み込むために、2 つのインスタンスを作成する。

```
train_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'train'))
test_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'test'))
```

```
train_imgs = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, 'train'))
test_imgs = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, 'test'))
```

最初の 8 個の正例と最後の 8 個の負例画像を以下に示す。見てわかるように、画像のサイズとアスペクト比はさまざまです。

```
hotdogs = [train_imgs[i][0] for i in range(8)]
not_hotdogs = [train_imgs[-i - 1][0] for i in range(8)]
d2l.show_images(hotdogs + not_hotdogs, 2, 8, scale=1.4);
```



訓練時には、まず画像からランダムなサイズとランダムなアスペクト比をもつ領域をランダムに切り出し、その領域を 224×224 の入力画像に拡大縮小する。テスト時には、画像の高さと幅の両方を 256 ピクセルに拡大縮小し、その後、中央の 224×224 領域を入力として切り出す。さらに、3 つの RGB（赤、緑、青）色チャネルについて、チャネルごとに値を 標準化する。具体的には、各チャネルの平均値をそのチャネルの各値から引き、その結果をそのチャネルの標準偏差で割る。

```
# Specify the means and standard deviations of the three RGB channels to
# standardize each channel
```

(continues on next page)

(continued from previous page)

```
normalize = torchvision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomResizedCrop(224),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    normalize])

test_augs = torchvision.transforms.Compose([
    torchvision.transforms.Resize([256, 256]),
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    normalize])
```

```
# Specify the means and standard deviations of the three RGB channels to
# standardize each channel
normalize = gluon.data.vision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.RandomResizedCrop(224),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    gluon.data.vision.transforms.ToTensor(),
    normalize])

test_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.Resize(256),
    gluon.data.vision.transforms.CenterCrop(224),
    gluon.data.vision.transforms.ToTensor(),
    normalize])
```

モデルの定義と初期化

ソースモデルとして、ImageNet データセットで事前学習された ResNet-18 を使用する。ここでは、事前学習済みモデルのパラメータを自動的にダウンロードするために pretrained=True を指定する。このモデルを初めて使用する場合は、ダウンロードのためにインターネット接続が必要である。

```
pretrained_net = torchvision.models.resnet18(pretrained=True)
```

```
pretrained_net = gluon.model_zoo.vision.resnet18_v2(pretrained=True)
```

```
Downloading /opt/mxnet/models/resnet18_v2-a81db45f.zip4c2e928f-7c57-4687-b3d7-
↪5d55c0413d9d from https://apache-mxnet.s3-accelerate.dualstack.amazonaws.com/
↪gluon/models/resnet18_v2-a81db45f.zip...
```

事前学習済みのソースモデルのインスタンスには、複数の特徴抽出層と出力層

fc がある。この分割の主な目的は、出力層以外のすべての層のモデルパラメータをファインチューニングしやすくすることである。ソースモデルのメンバー変数 fc を以下に示す。

```
pretrained_net.fc
```

```
Linear(in_features=512, out_features=1000, bias=True)
```

```
pretrained_net.output
```

```
Dense(512 -> 1000, linear)
```

全結合層として、ResNet の最後のグローバル平均プーリングの出力を ImageNet データセットの 1000 クラス出力に変換する。次に、ターゲットモデルとして新しいニューラルネットワークを構築する。これは事前学習済みのソースモデルと同じ方法で定義されるが、最終層の出力数だけがターゲットデータセットのクラス数（1000 ではなく）に設定される。

以下のコードでは、ターゲットモデルインスタンス finetune_net の出力層より前のモデルパラメータを、ソースモデルの対応する層のモデルパラメータで初期化する。これらのモデルパラメータは ImageNet 上での事前学習によって得られたものなので、有効である。したがって、こうした事前学習済みパラメータの ファインチューニングには、小さな学習率だけを使えば十分である。一方、出力層のモデルパラメータはランダムに初期化されるため、通常はゼロから学習するためにより大きな学習率が必要である。基準学習率を η とすると、出力層のモデルパラメータを更新する際には 10η の学習率を用いる。

```
finetune_net = torchvision.models.resnet18(pretrained=True)
finetune_net.fc = nn.Linear(finetune_net.fc.in_features, 2)
nn.init.xavier_uniform_(finetune_net.fc.weight);
```

```
finetune_net = gluon.model_zoo.vision.resnet18_v2(classes=2)
finetune_net.features = pretrained_net.features
finetune_net.output.initialize(init.Xavier())
# The model parameters in the output layer will be iterated using a learning
# rate ten times greater
finetune_net.output.collect_params().setattr('lr_mult', 10)
```

モデルのファインチューニング

まず、ファインチューニングを用いる訓練関数 train_fine_tuning を定義し、複数回呼び出せるようにする。

```
# If `param_group=True`, the model parameters in the output layer will be
# updated using a learning rate ten times greater
def train_fine_tuning(net, learning_rate, batch_size=128, num_epochs=5,
                      param_group=True):
    train_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
        os.path.join(data_dir, 'train'), transform=train_augs),
        batch_size=batch_size, shuffle=True)
    test_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
        os.path.join(data_dir, 'test'), transform=test_augs),
        batch_size=batch_size)
    devices = d2l.try_all_gpus()
    loss = nn.CrossEntropyLoss(reduction="none")
    if param_group:
        params_1x = [param for name, param in net.named_parameters()
             if name not in ["fc.weight", "fc.bias"]]
        trainer = torch.optim.SGD([{'params': params_1x},
                                   {'params': net.fc.parameters(),
                                    'lr': learning_rate * 10}],
                                  lr=learning_rate, weight_decay=0.001)
    else:
        trainer = torch.optim.SGD(net.parameters(), lr=learning_rate,
                                  weight_decay=0.001)
    d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
                   devices)
```

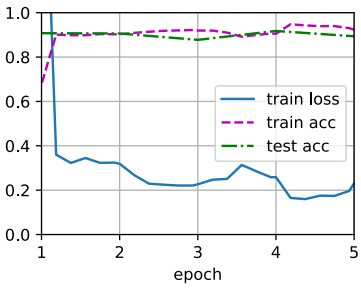
```
def train_fine_tuning(net, learning_rate, batch_size=128, num_epochs=5):
    train_iter = gluon.data.DataLoader(
        train_imgs.transform_first(train_augs), batch_size, shuffle=True)
    test_iter = gluon.data.DataLoader(
        test_imgs.transform_first(test_augs), batch_size)
    devices = d2l.try_all_gpus()
    net.collect_params().reset_ctx(devices)
    net.hybridize()
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {
        'learning_rate': learning_rate, 'wd': 0.001})
    d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
                   devices)
```

事前学習で得られたモデルパラメータをファインチューニングするために、基準学習率を小さな値に設定する。前述の設定に基づき、ターゲットモデルの出力層パラメータは、10 倍大きい学習率を用いてゼロから学習する。

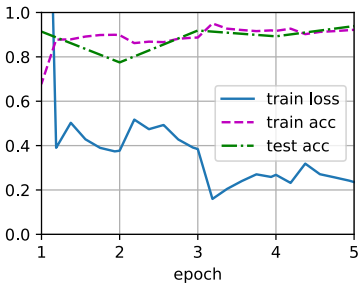
```
train_fine_tuning(finetune_net, 5e-5)
```

```
loss 0.231, train acc 0.923, test acc 0.894
1099.6 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```

```
train_fine_tuning(finetune_net, 0.01)
```



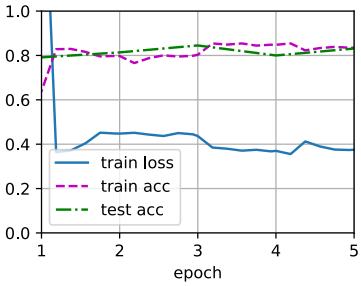

```
loss 0.236, train acc 0.922, test acc 0.939
369.4 examples/sec on [gpu(0), gpu(1)]
```



比較のために、同一のモデルを定義するが、すべてのモデルパラメータをランダム値で初期化する。モデル全体をゼロから学習する必要があるため、より大きな学習率を使うことができる。

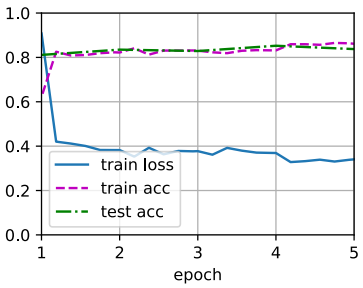
```
scratch_net = torchvision.models.resnet18()
scratch_net.fc = nn.Linear(scratch_net.fc.in_features, 2)
train_fine_tuning(scratch_net, 5e-4, param_group=False)
```

```
loss 0.375, train acc 0.835, test acc 0.831
1537.9 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



```
scratch_net = gluon.model_zoo.vision.resnet18_v2(classes=2)
scratch_net.initialize(init=init.Xavier())
train_fine_tuning(scratch_net, 0.1)
```

```
loss 0.340, train acc 0.862, test acc 0.838
304.8 examples/sec on [gpu(0), gpu(1)]
```



見てわかるように、ファインチューニングしたモデルは、初期パラメータ値がより有効であるため、同じエポック数ではより良い性能を示す傾向がある。

17.2.3 まとめ

- 転移学習は、ソースデータセットで学習した知識をターゲットデータセットへ転移する。ファインチューニングは転移学習でよく使われる手法である。
- ターゲットモデルは、出力層を除いてソースモデルからすべてのモデル設計とそのパラメータをコピーし、ターゲットデータセットに基づいてこれらのパラメータをファインチューニングする。一方、ターゲットモデルの出力層はゼロから学習する必要がある。
- 一般に、パラメータのファインチューニングには小さな学習率を使い、出力層をゼロから学習する場合にはより大きな学習率を使える。

17.2.4 演習

1. `finetune_net` の学習率をさらに大きくしていきなさい。モデルの精度はどのように変化するか？
2. 比較実験において、`finetune_net` と `scratch_net` のハイパーパラメータをさらに調整しなさい。それでも精度に差はあるか？
3. `finetune_net` の出力層より前のパラメータをソースモデルのものに設定し、訓練中にそれらを 更新しないようにしなさい。モデルの精度はどのように変化するか？次のコードを使える。

```
for param in finetune_net.parameters():
    param.requires_grad = False
```

```
finetune_net.features.collect_params().setattr('grad_req', 'null')
```

4. 実際には、ImageNet データセットには“hotdog”クラスがある。出力層における対応する重みパラメータは、次のコードで取得できる。この重みパラメータをどのように活用できるだろうか。

```
weight = pretrained_net.fc.weight
hotdog_w = torch.split(weight.data, 1, dim=0)[934]
hotdog_w.shape
```

```
torch.Size([1, 512])
```

```
weight = pretrained_net.output.weight
hotdog_w = np.split(weight.data(), 1000, axis=0)[713]
hotdog_w.shape
```

```
(1, 512)
```

17.3 物体検出とバウンディングボックス

前の節（たとえば Section 11.1–Section 11.4）では、画像分類のためのさまざまなモデルを紹介した。画像分類タスクでは、画像中に主要な物体がただ *1* つだけ存在すると仮定し、そのカテゴリを認識することだけに注目する。しかし、関心のある画像にはしばしば 複数の物体が含まれる。その場合、私たちはそれらのカテゴリだけでなく、画像内での具体的な位置も知りたいのである。コンピュータビジョンでは、このようなタスクを 物体検出（object detection、または 物体認識）と呼ぶ。

物体検出は、多くの分野で広く応用されている。たとえば、自動運転では、撮影した動画画像から車両、歩行者、道路、障害物の位置を検出することで、走行経路を計画する必要がある。また、ロボットはこの技術を用いて、環境内を移動する間に関心のある物体を検出し、その位置を特定することがある。さらに、セキュリティシステムでは、侵入者や爆弾のような異常な物体を検出する必要があるかもしれない。

次のいくつかの節では、物体検出のためのいくつかの深層学習手法を紹介する。まずは、物体の 位置（または 場所）の紹介から始める。

```
%matplotlib inline
from d2l import torch as d2l
import torch
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import image, npx, np

npx.set_np()
```

```
#@save
def box_corner_to_center(boxes):
    """Convert from (upper-left, lower-right) to (center, width, height)."""
    x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
    h = y2 - y1
    boxes = d2l.stack((cx, cy, w, h), axis=-1)
    return boxes

#@save
def box_center_to_corner(boxes):
    """Convert from (center, width, height) to (upper-left, lower-right)."""
    cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    x1 = cx - 0.5 * w
    y1 = cy - 0.5 * h
    x2 = cx + 0.5 * w
    y2 = cy + 0.5 * h
    boxes = d2l.stack((x1, y1, x2, y2), axis=-1)
    return boxes
```

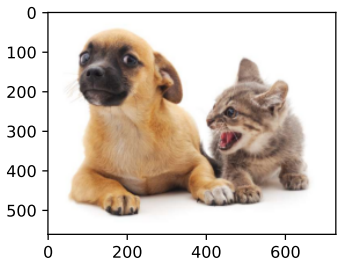
```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf
```

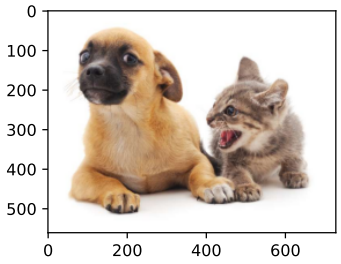
この節で使用するサンプル画像を読み込みる。画像の左側に犬、右側に猫がいることがわかる。これらがこの画像における 2 つの主要な物体である。

```
d2l.set_figsize()
img = d2l.plt.imread('../img/catdog.jpg')
d2l.plt.imshow(img);
```

```
d2l.set_figsize()
img = image.imread('../img/catdog.jpg').asnumpy()
d2l.plt.imshow(img);
```

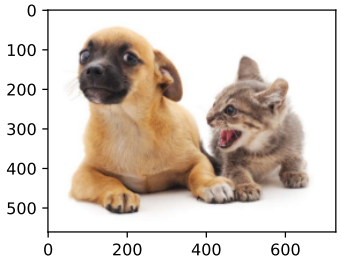
```
[07:02:09] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```






```
# Here `bbox` is the abbreviation for bounding box
dog_bbox, cat_bbox = [60.0, 45.0, 378.0, 516.0], [400.0, 112.0, 655.0, 493.0]
```

```
d2l.set_figsize()
img = d2l.plt.imread('../img/catdog.jpg')
d2l.plt.imshow(img);
```



17.3.1 バウンディングボックス

物体検出では、通常、物体の空間的位置を表すために バウンディングボックス を用いる。バウンディングボックスは長方形であり、長方形の左上隅の x 座標と y 座標、および右下隅の同様の座標によって決まりる。もう 1 つの一般的なバウンディングボックス表現は、バウンディングボックス中心の (x, y) 座標と、ボックスの幅および高さである。

ここでは、これら 2 つの表現の間を変換する関数を定義する。`box_corner_to_center` は 2 隅表現から中心・幅・高さ表現へ変換し、`box_center_to_corner` はその逆

を行う。入力引数 boxes は形状 $(n, 4)$ の 2 次元テンソルである必要があり、ここで n はバウンディングボックスの数である。

```
#@save
def box_corner_to_center(boxes):
    """Convert from (upper-left, lower-right) to (center, width, height)."""
    x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
    h = y2 - y1
    boxes = d2l.stack((cx, cy, w, h), axis=-1)
    return boxes

#@save
def box_center_to_corner(boxes):
    """Convert from (center, width, height) to (upper-left, lower-right)."""
    cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    x1 = cx - 0.5 * w
    y1 = cy - 0.5 * h
    x2 = cx + 0.5 * w
    y2 = cy + 0.5 * h
    boxes = d2l.stack((x1, y1, x2, y2), axis=-1)
    return boxes
```


```
#@save
def box_corner_to_center(boxes):
    """Convert from (upper-left, lower-right) to (center, width, height)."""
    x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
    h = y2 - y1
    boxes = d2l.stack((cx, cy, w, h), axis=-1)
    return boxes

#@save
def box_center_to_corner(boxes):
    """Convert from (center, width, height) to (upper-left, lower-right)."""
    cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    x1 = cx - 0.5 * w
    y1 = cy - 0.5 * h
    x2 = cx + 0.5 * w
    y2 = cy + 0.5 * h
    boxes = d2l.stack((x1, y1, x2, y2), axis=-1)
    return boxes
```


```
boxes = d2l.tensor((dog_bbox, cat_bbox))
box_center_to_corner(box_corner_to_center(boxes)) == boxes
```

```
#@save
def box_corner_to_center(boxes):
    """Convert from (upper-left, lower-right) to (center, width, height)."""
```

(continues on next page)

(continued from previous page)

```
    x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
    h = y2 - y1
    boxes = d2l.stack((cx, cy, w, h), axis=-1)
    return boxes

#@save
def box_center_to_corner(boxes):
    """Convert from (center, width, height) to (upper-left, lower-right)."""
    cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    x1 = cx - 0.5 * w
    y1 = cy - 0.5 * h
    x2 = cx + 0.5 * w
    y2 = cy + 0.5 * h
    boxes = d2l.stack((x1, y1, x2, y2), axis=-1)
    return boxes
```

座標情報に基づいて、画像中の犬と猫のバウンディングボックスを定義する。画像における座標の原点は画像の左上隅であり、右方向と下方向がそれぞれ x 軸と y 軸の正の方向である。

```
# Here `bbox` is the abbreviation for bounding box
dog_bbox, cat_bbox = [60.0, 45.0, 378.0, 516.0], [400.0, 112.0, 655.0, 493.0]
```


```
# Here `bbox` is the abbreviation for bounding box
dog_bbox, cat_bbox = [60.0, 45.0, 378.0, 516.0], [400.0, 112.0, 655.0, 493.0]
```


```
#@save
def bbox_to_rect(bbox, color):
    """Convert bounding box to matplotlib format."""
    # Convert the bounding box (upper-left x, upper-left y, lower-right x,
    # lower-right y) format to the matplotlib format: ((upper-left x,
    # upper-left y), width, height)
    return d2l.plt.Rectangle(
        xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],
        fill=False, edgecolor=color, linewidth=2)
```


```
# Here `bbox` is the abbreviation for bounding box
dog_bbox, cat_bbox = [60.0, 45.0, 378.0, 516.0], [400.0, 112.0, 655.0, 493.0]
```


2 回変換することで、2 つのバウンディングボックス変換関数の正しさを確認できる。

```
boxes = d2l.tensor((dog_bbox, cat_bbox))
box_center_to_corner(box_corner_to_center(boxes)) == boxes
```

```
tensor([[True, True, True, True],
        [True, True, True, True]])
```

```
boxes = d2l.tensor((dog_bbox, cat_bbox))
box_center_to_corner(box_corner_to_center(boxes)) == boxes
```

```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

```
fig = d2l.plt.imshow(img)
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```

```
boxes = d2l.tensor((dog_bbox, cat_bbox))
box_center_to_corner(box_corner_to_center(boxes)) == boxes
```

```
<tf.Tensor: shape=(2, 4), dtype=bool, numpy=
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True]])>
```

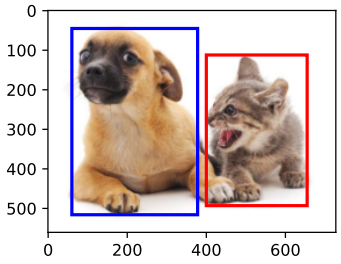
正確に描けているか確認するために、画像にバウンディングボックスを描画してみましょう。描画する前に、補助関数 bbox_to_rect を定義する。これはバウンディングボックスを matplotlib パッケージのバウンディングボックス形式で表す。

```
#@save
def bbox_to_rect(bbox, color):
    """Convert bounding box to matplotlib format."""
    # Convert the bounding box (upper-left x, upper-left y, lower-right x,
    # lower-right y) format to the matplotlib format: ((upper-left x,
    # upper-left y), width, height)
    return d2l.plt.Rectangle(
        xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],
        fill=False, edgecolor=color, linewidth=2)
```

画像にバウンディングボックスを追加すると、2 つの物体の主な輪郭が、基本的に 2 つのボックスの内側に収まっていることがわかる。

```
fig = d2l.plt.imshow(img)
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```

17.3.2 まとめ



- 物体検出では、画像中の関心のあるすべての物体を認識するだけでなく、それらの位置も認識する。位置は一般に長方形のバウンディングボックスで表される。
- よく使われる 2 種類のバウンディングボックス表現の間で変換できる。

17.3.3 演習

1. 別の画像を見つけて、物体を含むバウンディングボックスをラベル付けしてみなさい。バウンディングボックスのラベル付けとカテゴリのラベル付けを比較すると、通常どちらに時間がかかりますか？
2. box_corner_to_center と box_center_to_corner の入力引数 boxes の最内次元が常に 4 であるのはなぜですか？

17.4 アンカーボックス

物体検出アルゴリズムは通常、入力画像内の多数の領域をサンプリングし、それらの領域に注目対象の物体が含まれているかを判定し、さらに領域の境界を調整して、物体の 真のバウンディングボックスをより正確に予測する。モデルによって、領域のサンプリング方法はさまざまである。ここではそのような方法の一つを紹介する。これは、各ピクセルを中心として、異なるスケールとアスペクト比をもつ複数のバウンディングボックスを生成する。これらのバウンディングボックスを アンカーボックスと呼ぶ。Section 17.7 では、アンカーボックスに基づく物体検出モデルを設計する。

まず、出力をより簡潔にするために、表示精度を少し変更しよう。

```
%matplotlib inline
from d2l import torch as d2l
import torch

torch.set_printoptions(2)  # 表示精度を簡略化
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, np, npx

np.set_printoptions(2)  # 表示精度を簡略化
npx.set_np()
```

```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```


```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```


17.4.1 複数のアンカーボックスの生成

入力画像の高さが h、幅が w であるとする。画像の各ピクセルを中心として、異なる形状のアンカーボックスを生成する。スケールを $s \in (0,1]$、アスペクト比（幅と高さの比）を $r > 0$ とする。すると、アンカーボックスの幅と高さはそれぞれ $ws\sqrt{r}$ と $hs/\sqrt{r}$ になる。中心位置が与えられれば、幅と高さが既知のアンカーボックスは一意に定まることに注意しよ。

異なる形状の複数のアンカーボックスを生成するために、一連のスケール $s_1, \ldots, s_n$ と一連のアスペクト比 $r_1, \ldots, r_m$ を設定する。各ピクセルを中心として、これらのスケールとアスペクト比の全組合せを用いると、入力画像には合計 $whnm$ 個のアンカーボックスが生成される。これらのアンカーボックスはすべての真のバウンディングボックスを覆えるかもしれないが、計算量が容易に高くなりすぎる。実際には、s_1 または r_1 を含む組合せだけを考慮することができる。

$$(s_1, r_1), (s_1, r_2), \ldots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \ldots, (s_n, r_1). \tag{17.4.1}$$

つまり、同じピクセルを中心とするアンカーボックスの数は $n+m-1$ である。入力画像全体では、合計 $wh(n+m-1)$ 個のアンカーボックスを生成する。

上記のアンカーボックス生成方法は、次の `multibox_prior` 関数で実装されている。入力画像、スケールのリスト、アスペクト比のリストを指定すると、この関数はすべてのアンカーボックスを返す。

```
#@save
def multibox_prior(data, sizes, ratios):
    """Generate anchor boxes with different shapes centered on each pixel."""
    in_height, in_width = data.shape[-2:]
    device, num_sizes, num_ratios = data.device, len(sizes), len(ratios)
    boxes_per_pixel = (num_sizes + num_ratios - 1)
    size_tensor = d2l.tensor(sizes, device=device)
    ratio_tensor = d2l.tensor(ratios, device=device)
    # Offsets are required to move the anchor to the center of a pixel. Since
```

(continues on next page)

(continued from previous page)

```
    # a pixel has height=1 and width=1, we choose to offset our centers by 0.5
    offset_h, offset_w = 0.5, 0.5
    steps_h = 1.0 / in_height  # Scaled steps in y axis
    steps_w = 1.0 / in_width  # Scaled steps in x axis

    # Generate all center points for the anchor boxes
    center_h = (torch.arange(in_height, device=device) + offset_h) * steps_h
    center_w = (torch.arange(in_width, device=device) + offset_w) * steps_w
    shift_y, shift_x = torch.meshgrid(center_h, center_w, indexing='ij')
    shift_y, shift_x = shift_y.reshape(-1), shift_x.reshape(-1)

    # Generate `boxes_per_pixel` number of heights and widths that are later
    # used to create anchor box corner coordinates (xmin, xmax, ymin, ymax)
    w = torch.cat((size_tensor * torch.sqrt(ratio_tensor[0]),
                   sizes[0] * torch.sqrt(ratio_tensor[1:])))\
                   * in_height / in_width  # Handle rectangular inputs
    h = torch.cat((size_tensor / torch.sqrt(ratio_tensor[0]),
                   sizes[0] / torch.sqrt(ratio_tensor[1:])))
    # Divide by 2 to get half height and half width
    anchor_manipulations = torch.stack((-w, -h, w, h)).T.repeat(
                                        in_height * in_width, 1) / 2

    # Each center point will have `boxes_per_pixel` number of anchor boxes, so
    # generate a grid of all anchor box centers with `boxes_per_pixel` repeats
    out_grid = torch.stack([shift_x, shift_y, shift_x, shift_y],
                dim=1).repeat_interleave(boxes_per_pixel, dim=0)
    output = out_grid + anchor_manipulations
    return output.unsqueeze(0)
```

```
#@save
def multibox_prior(data, sizes, ratios):
    """Generate anchor boxes with different shapes centered on each pixel."""
    in_height, in_width = data.shape[-2:]
    device, num_sizes, num_ratios = data.ctx, len(sizes), len(ratios)
    boxes_per_pixel = (num_sizes + num_ratios - 1)
    size_tensor = d2l.tensor(sizes, ctx=device)
    ratio_tensor = d2l.tensor(ratios, ctx=device)
    # Offsets are required to move the anchor to the center of a pixel. Since
    # a pixel has height=1 and width=1, we choose to offset our centers by 0.5
    offset_h, offset_w = 0.5, 0.5
    steps_h = 1.0 / in_height  # Scaled steps in y-axis
    steps_w = 1.0 / in_width  # Scaled steps in x-axis

    # Generate all center points for the anchor boxes
    center_h = (d2l.arange(in_height, ctx=device) + offset_h) * steps_h
    center_w = (d2l.arange(in_width, ctx=device) + offset_w) * steps_w
    shift_x, shift_y = d2l.meshgrid(center_w, center_h)
    shift_x, shift_y = shift_x.reshape(-1), shift_y.reshape(-1)

    # Generate `boxes_per_pixel` number of heights and widths that are later
    # used to create anchor box corner coordinates (xmin, xmax, ymin, ymax)
    w = np.concatenate((size_tensor * np.sqrt(ratio_tensor[0]),
                        sizes[0] * np.sqrt(ratio_tensor[1:])) \
                        * in_height / in_width  # Handle rectangular inputs
```

(continues on next page)

(continued from previous page)

```
    h = np.concatenate((size_tensor / np.sqrt(ratio_tensor[0]),
                        sizes[0] / np.sqrt(ratio_tensor[1:])))
    # Divide by 2 to get half height and half width
    anchor_manipulations = np.tile(np.stack((-w, -h, w, h)).T,
                                   (in_height * in_width, 1)) / 2

    # Each center point will have `boxes_per_pixel` number of anchor boxes, so
    # generate a grid of all anchor box centers with `boxes_per_pixel` repeats
    out_grid = d2l.stack([shift_x, shift_y, shift_x, shift_y],
                         axis=1).repeat(boxes_per_pixel, axis=0)
    output = out_grid + anchor_manipulations
    return np.expand_dims(output, axis=0)
```

```
#@save
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """Show bounding boxes."""

    def make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj

    labels = make_list(labels)
    colors = make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(d2l.numpy(bbox), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va='center', ha='center', fontsize=9, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

```
#@save
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """Show bounding boxes."""

    def make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj

    labels = make_list(labels)
    colors = make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(d2l.numpy(bbox), color)
        axes.add_patch(rect)
```

(continues on next page)

(continued from previous page)

```
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va='center', ha='center', fontsize=9, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

返されるアンカーボックス変数 `Y` の形状は、(batch size, number of anchor boxes, 4) である。

```
img = d2l.plt.imread('../img/catdog.jpg')
h, w = img.shape[:2]

print(h, w)
X = torch.rand(size=(1, 3, h, w))  # Construct input data
Y = multibox_prior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

```
561 728
```

```
torch.Size([1, 2042040, 4])
```

```
img = image.imread('../img/catdog.jpg').asnumpy()
h, w = img.shape[:2]

print(h, w)
X = np.random.uniform(size=(1, 3, h, w))  # Construct input data
Y = multibox_prior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

```
561 728
[07:07:29] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(1, 2042040, 4)
```

```
d2l.set_figsize()
bbox_scale = d2l.tensor((w, h, w, h))
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
```

```
d2l.set_figsize()
bbox_scale = d2l.tensor((w, h, w, h))
fig = d2l.plt.imshow(img)
```

(continues on next page)

(continued from previous page)

```
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
```

アンカーボックス変数 Y の形状を (image height, image width, number of anchor boxes centered on the same pixel, 4) に変更すると、指定したピクセル位置を中心とするすべてのアンカーボックスを得ることができる。以下では、(250, 250) を中心とする最初のアンカーボックスを参照する。これは 4 つの要素を持つ。すなわち、アンカーボックスの左上隅の (x, y) 軸座標と右下隅の (x, y) 軸座標である。両軸の座標値は、それぞれ画像の幅と高さで割られている。

```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```

```
tensor([0.06, 0.07, 0.63, 0.82])
```

```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```

```
array([0.06, 0.07, 0.63, 0.82])
```

```
#@save
def offset_boxes(anchors, assigned_bb, eps=1e-6):
    """Transform for anchor box offsets."""
    c_anc = d2l.box_corner_to_center(anchors)
    c_assigned_bb = d2l.box_corner_to_center(assigned_bb)
    offset_xy = 10 * (c_assigned_bb[:, :2] - c_anc[:, :2]) / c_anc[:, 2:]
    offset_wh = 5 * d2l.log(eps + c_assigned_bb[:, 2:] / c_anc[:, 2:])
    offset = d2l.concat([offset_xy, offset_wh], axis=1)
    return offset
```

```
#@save
def offset_boxes(anchors, assigned_bb, eps=1e-6):
    """Transform for anchor box offsets."""
    c_anc = d2l.box_corner_to_center(anchors)
    c_assigned_bb = d2l.box_corner_to_center(assigned_bb)
    offset_xy = 10 * (c_assigned_bb[:, :2] - c_anc[:, :2]) / c_anc[:, 2:]
    offset_wh = 5 * d2l.log(eps + c_assigned_bb[:, 2:] / c_anc[:, 2:])
    offset = d2l.concat([offset_xy, offset_wh], axis=1)
    return offset
```

画像内の 1 つのピクセルを中心とするすべてのアンカーボックスを表示するために、複数のバウンディングボックスを画像上に描画する次の show_bboxes 関数を定義する。

```
#@save
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """Show bounding boxes."""

    def make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj

    labels = make_list(labels)
    colors = make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(d2l.numpy(bbox), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va='center', ha='center', fontsize=9, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

```
#@save
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """Show bounding boxes."""

    def make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj

    labels = make_list(labels)
    colors = make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(d2l.numpy(bbox), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va='center', ha='center', fontsize=9, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

```
ground_truth = d2l.tensor([[0, 0.1, 0.08, 0.52, 0.92],
                           [1, 0.55, 0.2, 0.9, 0.88]])
anchors = d2l.tensor([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                      [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                      [0.57, 0.3, 0.92, 0.9]])

fig = d2l.plt.imshow(img)
```

(continues on next page)

(continued from previous page)

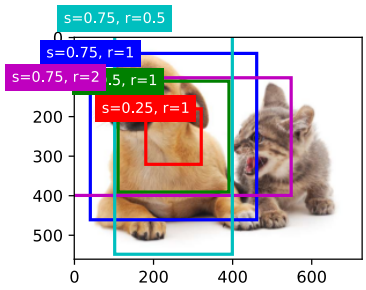
```
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```

```
ground_truth = d2l.tensor([[0, 0.1, 0.08, 0.52, 0.92],
                           [1, 0.55, 0.2, 0.9, 0.88]])
anchors = d2l.tensor([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                      [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                      [0.57, 0.3, 0.92, 0.9]])

fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```

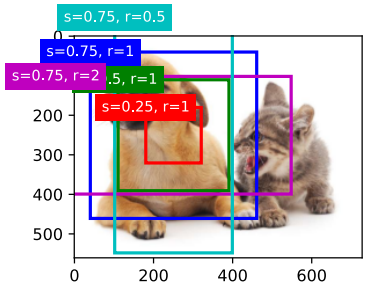
先ほど見たように、変数 boxes における x 軸と y 軸の座標値は、それぞれ画像の幅と高さで割られている。アンカーボックスを描画する際には、元の座標値に戻す必要がある。そのため、以下で変数 bbox_scale を定義する。これで、画像内の $(250, 250)$ を中心とするすべてのアンカーボックスを描画できる。ご覧のとおり、スケール 0.75、アスペクト比 1 の青いアンカーボックスは、画像中の犬をうまく囲んでいる。

```
d2l.set_figsize()
bbox_scale = d2l.tensor((w, h, w, h))
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
```




```
d2l.set_figsize()
bbox_scale = d2l.tensor((w, h, w, h))
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
```

```
labels[2]
```




```
labels[2]
```

17.4.2 Intersection over Union (IoU)

先ほど、アンカーボックスが画像中の犬を「うまく」囲んでいると述べた。物体の真のバウンディングボックスが分かっている場合、この「うまく」はどのように定量化できるだろうか。直感的には、アンカーボックスと真のバウンディングボックスの類似度を測れる。*Jaccard index* は 2 つの集合の類似度を測ることができる。集合 $\mathcal{A}$ と $\mathcal{B}$ に対して、その Jaccard index は、共通部分の大きさを和集合の大きさで割ったものである。

$$J(\mathcal{A}, \mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}. \tag{17.4.2}$$

実際には、任意のバウンディングボックスのピクセル領域をピクセルの集合として考えることができる。このようにして、2 つのバウンディングボックスの類似度を、それらのピクセル集合の Jaccard index によって測定できる。2 つのバウンディングボックスについて、通常はその Jaccard index を *intersection over union*（*IoU*）と呼ぶ。これは、Fig. 17.4.1 に示すように、共通部分の面積を和集合の面積で割った比である。IoU の範囲は 0 から 1 である。0 は 2 つのバウンディングボックスがまったく重なっていないことを意味し、1 は 2 つのバウンディングボックスが等しいことを示す。

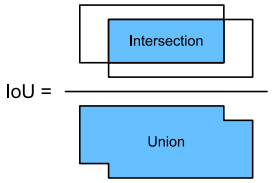


Fig. 17.4.1 IoU is the ratio of the intersection area to the union area of two bounding boxes.

この節の残りでは、IoU を用いてアンカーボックスと真のバウンディングボックスの類似度、および異なるアンカーボックス同士の類似度を測りる。2 つのアンカーまたはバウンディングボックスのリストが与えられたとき、次の `box_iou` は、これら 2 つのリスト間の全組合せについて IoU を計算する。

```
#@save
def box_iou(boxes1, boxes2):
    """Compute pairwise IoU across two lists of anchor or bounding boxes."""
    box_area = lambda boxes: ((boxes[:, 2] - boxes[:, 0]) *
                              (boxes[:, 3] - boxes[:, 1]))
    # Shape of `boxes1`, `boxes2`, `areas1`, `areas2`: (no. of boxes1, 4),
    # (no. of boxes2, 4), (no. of boxes1,), (no. of boxes2,)
    areas1 = box_area(boxes1)
    areas2 = box_area(boxes2)
    # Shape of `inter_upperlefts`, `inter_lowerrights`, `inters`: (no. of
    # boxes1, no. of boxes2, 2)
    inter_upperlefts = torch.max(boxes1[:, None, :2], boxes2[:, :2])
    inter_lowerrights = torch.min(boxes1[:, None, 2:], boxes2[:, 2:])
    inters = (inter_lowerrights - inter_upperlefts).clamp(min=0)
    # Shape of `inter_areas` and `union_areas`: (no. of boxes1, no. of boxes2)
    inter_areas = inters[:, :, 0] * inters[:, :, 1]
    union_areas = areas1[:, None] + areas2 - inter_areas
    return inter_areas / union_areas
```

```
#@save
def box_iou(boxes1, boxes2):
    """Compute pairwise IoU across two lists of anchor or bounding boxes."""
    box_area = lambda boxes: ((boxes[:, 2] - boxes[:, 0]) *
                              (boxes[:, 3] - boxes[:, 1]))
    # Shape of `boxes1`, `boxes2`, `areas1`, `areas2`: (no. of boxes1, 4),
    # (no. of boxes2, 4), (no. of boxes1,), (no. of boxes2,)
    areas1 = box_area(boxes1)
    areas2 = box_area(boxes2)
    # Shape of `inter_upperlefts`, `inter_lowerrights`, `inters`: (no. of
    # boxes1, no. of boxes2, 2)
    inter_upperlefts = np.maximum(boxes1[:, None, :2], boxes2[:, :2])
    inter_lowerrights = np.minimum(boxes1[:, None, 2:], boxes2[:, 2:])
    inters = (inter_lowerrights - inter_upperlefts).clip(min=0)
    # Shape of `inter_areas` and `union_areas`: (no. of boxes1, no. of boxes2)
    inter_areas = inters[:, :, 0] * inters[:, :, 1]
    union_areas = areas1[:, None] + areas2 - inter_areas
    return inter_areas / union_areas
```

```
labels[1]
```

```
labels[1]
```

17.4.3 学習データにおけるアンカーボックスのラベル付け

学習データセットでは、各アンカーボックスを 1 つの学習例とみなす。物体検出モデルを学習するためには、各アンカーボックスに対して クラスラベルと オフセットラベルが必要である。前者はアンカーボックスに対応する物体のクラスであり、後者はアンカーボックスに対する真のバウンディングボックスのオフセットである。予測時には、各画像について複数のアンカーボックスを生成し、すべてのアンカーボックスについてクラスとオフセットを予測し、

予測されたオフセットに従って位置を調整して予測バウンディングボックスを得て、最後に一定の条件を満たす予測バウンディングボックスだけを出力する。

ご存じのように、物体検出の学習データセットには、真のバウンディングボックスの位置と、それらに囲まれた物体のクラスのラベルが付いている。生成された任意の アンカーボックスにラベルを付けるには、そのアンカーボックスに最も近い 割り当てられた真のバウンディングボックスのラベル付き位置とクラスを参照する。以下では、アンカーボックスに最も近い真のバウンディングボックスを割り当てるアルゴリズムを説明する。

真のバウンディングボックスをアンカーボックスに割り当てる

画像が与えられたとき、アンカーボックスを $A_1, A_2, \ldots, A_{n_a}$、真のバウンディングボックスを $B_1, B_2, \ldots, B_{n_b}$ とし、ここで $n_a \geq n_b$ とする。行列 $\mathbf{X} \in \mathbb{R}^{n_a \times n_b}$ を定義し、その i^{th} 行 j^{th} 列の要素 x_{ij} をアンカーボックス A_i と真のバウンディングボックス B_j の IoU とする。アルゴリズムは次の手順からなる。

1. 行列 $\mathbf{X}$ の最大要素を見つけ、その行と列のインデックスをそれぞれ i_1 と j_1 とする。すると、真のバウンディングボックス B_{j_1} はアンカーボックス A_{i_1} に割り当てられる。これは、すべてのアンカーボックスと真のバウンディングボックスの組の中で A_{i_1} と B_{j_1} が最も近いからであり、直感的である。最初の割り当ての後、行列 $\mathbf{X}$ の ${i_1}^{\text{th}}$ 行と ${j_1}^{\text{th}}$ 列のすべての要素を破棄する。

2. 行列 $\mathbf{X}$ に残っている要素の最大値を見つけ、その行と列のインデックスをそれぞれ i_2 と j_2 とする。真のバウンディングボックス B_{j_2} をアンカーボックス A_{i_2} に割り当て、行列 $\mathbf{X}$ の ${i_2}^{\text{th}}$ 行と ${j_2}^{\text{th}}$ 列のすべての要素を破棄する。

3. この時点で、行列 $\mathbf{X}$ の 2 つの行と 2 つの列の要素が破棄されている。行列 $\mathbf{X}$ の n_b 列すべての要素が破棄されるまで続ける。この時点で、n_b 個のアンカーボックスそれぞれに真のバウンディングボックスが割り当てられている。

4. 残りの $n_a - n_b$ 個のアンカーボックスだけを走査する。たとえば、任意のアンカーボックス A_i について、行列 $\mathbf{X}$ の i^{th} 行全体で A_i と最も IoU が大きい真のバウンディングボックス B_j を見つけ、この IoU があらかじめ定めたしきい値より大きい場合にのみ B_j を A_i に割り当てる。

上記のアルゴリズムを具体例で説明しよう。Fig. 17.4.2（左）に示すように、行列 $\mathbf{X}$ の最大値が x_{23} だとすると、真のバウンディングボックス B_3 をアンカーボックス A_2 に割り当てる。次に、行 2 と列 3 のすべての要素を破棄し、残りの要素（網掛け部分）の中で最大の x_{71} を見つけ、真のバウンディングボックス B_1 をアンカーボックス A_7 に割り当てる。次に、Fig. 17.4.2（中央）に示すように、行 7 と列 1 のすべての要素を破棄し、残りの要素（網掛け部分）の中で最大の x_{54} を見つけ、真のバウンディングボックス B_4 をアンカーボックス A_5 に割り当てる。最後に、Fig. 17.4.2（右）に示すように、行 5 と列 4 のすべての要素を破棄し、残りの要素（網掛け部分）の中で最大の x_{92} を見つけ、真のバウンディングボックス B_2 をアンカーボックス A_9 に割り当てる。その後は、残りのアンカ

ーボックス A_1, A_3, A_4, A_6, A_8 を走査し、しきい値に従って真のバウンディングボックスを割り当てるかどうかを決めるだけである。

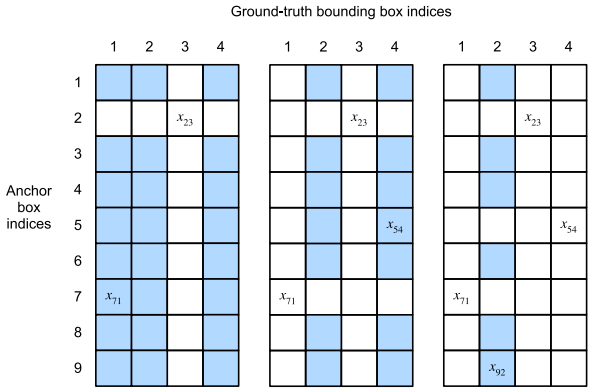


Fig. 17.4.2 Assigning ground-truth bounding boxes to anchor boxes.

このアルゴリズムは、次の assign_anchor_to_bbox 関数で実装されている。

```
#@save
def assign_anchor_to_bbox(ground_truth, anchors, device, iou_threshold=0.5):
    """Assign closest ground-truth bounding boxes to anchor boxes."""
    num_anchors, num_gt_boxes = anchors.shape[0], ground_truth.shape[0]
    # Element x_ij in the i-th row and j-th column is the IoU of the anchor
    # box i and the ground-truth bounding box j
    jaccard = box_iou(anchors, ground_truth)
    # Initialize the tensor to hold the assigned ground-truth bounding box for
    # each anchor
    anchors_bbox_map = torch.full((num_anchors,), -1, dtype=torch.long,
                                  device=device)
    # Assign ground-truth bounding boxes according to the threshold
    max_ious, indices = torch.max(jaccard, dim=1)
    anc_i = torch.nonzero(max_ious >= iou_threshold).reshape(-1)
    box_j = indices[max_ious >= iou_threshold]
    anchors_bbox_map[anc_i] = box_j
    col_discard = torch.full((num_anchors,), -1)
    row_discard = torch.full((num_gt_boxes,), -1)
    for _ in range(num_gt_boxes):
        max_idx = torch.argmax(jaccard)  # Find the largest IoU
        box_idx = (max_idx % num_gt_boxes).long()
        anc_idx = (max_idx / num_gt_boxes).long()
        anchors_bbox_map[anc_idx] = box_idx
        jaccard[:, box_idx] = col_discard
        jaccard[anc_idx, :] = row_discard
    return anchors_bbox_map
```

```
#@save
def assign_anchor_to_bbox(ground_truth, anchors, device, iou_threshold=0.5):
    """Assign closest ground-truth bounding boxes to anchor boxes."""
    num_anchors, num_gt_boxes = anchors.shape[0], ground_truth.shape[0]
    # Element x_ij in the i-th row and j-th column is the IoU of the anchor
```

(continues on next page)

(continued from previous page)

```
    # box i and the ground-truth bounding box j
    jaccard = box_iou(anchors, ground_truth)
    # Initialize the tensor to hold the assigned ground-truth bounding box for
    # each anchor
    anchors_bbox_map = np.full((num_anchors,), -1, dtype=np.int32, ctx=device)
    # Assign ground-truth bounding boxes according to the threshold
    max_ious, indices = np.max(jaccard, axis=1), np.argmax(jaccard, axis=1)
    anc_i = np.nonzero(max_ious >= iou_threshold)[0]
    box_j = indices[max_ious >= iou_threshold]
    anchors_bbox_map[anc_i] = box_j
    col_discard = np.full((num_anchors,), -1)
    row_discard = np.full((num_gt_boxes,), -1)
    for _ in range(num_gt_boxes):
        max_idx = np.argmax(jaccard)  # Find the largest IoU
        box_idx = (max_idx % num_gt_boxes).astype('int32')
        anc_idx = (max_idx / num_gt_boxes).astype('int32')
        anchors_bbox_map[anc_idx] = box_idx
        jaccard[:, box_idx] = col_discard
        jaccard[anc_idx, :] = row_discard
    return anchors_bbox_map
```

```
labels[0]
```

```
labels[0]
```

クラスとオフセットのラベル付け

これで、各アンカーボックスのクラスとオフセットにラベルを付けることができる。アンカーボックス A に真のバウンディングボックス B が割り当てられたとする。一方で、アンカーボックス A のクラスは B のクラスとしてラベル付けされる。他方で、アンカーボックス A のオフセットは、B と A の中心座標の相対位置と、これら 2 つのボックスの相対サイズに基づいてラベル付けされる。データセット内のさまざまなボックスの位置とサイズが異なるため、これらの相対位置とサイズに変換を施すことで、より一様に分布したオフセットを得られ、学習しやすくなる。ここでは一般的な変換を説明する。A と B の中心座標をそれぞれ (x_a, y_a) と (x_b, y_b)、幅を w_a と w_b、高さを h_a と h_b とする。このとき、A のオフセットは次のようにラベル付けできる。

$$\left(\frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log\frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log\frac{h_b}{h_a} - \mu_h}{\sigma_h}\right), \tag{17.4.3}$$

ここで定数のデフォルト値は $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1$, および $\sigma_w = \sigma_h = 0.2$ である。この変換は以下の `offset_boxes` 関数で実装されている。

```
#@save
def offset_boxes(anchors, assigned_bb, eps=1e-6):
```

(continues on next page)

(continued from previous page)

```
    """Transform for anchor box offsets."""
    c_anc = d2l.box_corner_to_center(anchors)
    c_assigned_bb = d2l.box_corner_to_center(assigned_bb)
    offset_xy = 10 * (c_assigned_bb[:, :2] - c_anc[:, :2]) / c_anc[:, 2:]
    offset_wh = 5 * d2l.log(eps + c_assigned_bb[:, 2:] / c_anc[:, 2:])
    offset = d2l.concat([offset_xy, offset_wh], axis=1)
    return offset
```

```
#@save
def offset_boxes(anchors, assigned_bb, eps=1e-6):
    """Transform for anchor box offsets."""
    c_anc = d2l.box_corner_to_center(anchors)
    c_assigned_bb = d2l.box_corner_to_center(assigned_bb)
    offset_xy = 10 * (c_assigned_bb[:, :2] - c_anc[:, :2]) / c_anc[:, 2:]
    offset_wh = 5 * d2l.log(eps + c_assigned_bb[:, 2:] / c_anc[:, 2:])
    offset = d2l.concat([offset_xy, offset_wh], axis=1)
    return offset
```

```
#@save
def offset_inverse(anchors, offset_preds):
    """Predict bounding boxes based on anchor boxes with predicted offsets."""
    anc = d2l.box_corner_to_center(anchors)
    pred_bbox_xy = (offset_preds[:, :2] * anc[:, 2:] / 10) + anc[:, :2]
    pred_bbox_wh = d2l.exp(offset_preds[:, 2:] / 5) * anc[:, 2:]
    pred_bbox = d2l.concat((pred_bbox_xy, pred_bbox_wh), axis=1)
    predicted_bbox = d2l.box_center_to_corner(pred_bbox)
    return predicted_bbox
```

```
#@save
def offset_inverse(anchors, offset_preds):
    """Predict bounding boxes based on anchor boxes with predicted offsets."""
    anc = d2l.box_corner_to_center(anchors)
    pred_bbox_xy = (offset_preds[:, :2] * anc[:, 2:] / 10) + anc[:, :2]
    pred_bbox_wh = d2l.exp(offset_preds[:, 2:] / 5) * anc[:, 2:]
    pred_bbox = d2l.concat((pred_bbox_xy, pred_bbox_wh), axis=1)
    predicted_bbox = d2l.box_center_to_corner(pred_bbox)
    return predicted_bbox
```

アンカーボックスに真のバウンディングボックスが割り当てられていない場合、そのアンカーボックスのクラスは単に「背景」とラベル付けする。クラスが背景であるアンカーボックスはしばしば 負例アンカーボックスと呼ばれ、それ以外は 正例アンカーボックスと呼ばれる。以下の `multibox_target` 関数を実装して、真のバウンディングボックス（`labels` 引数）を用いて、アンカーボックス（`anchors` 引数）のクラスとオフセットにラベルを付けます。この関数では背景クラスを 0 とし、新しいクラスの整数インデックスは 1 から始める。

```
#@save
def multibox_target(anchors, labels):
    """Label anchor boxes using ground-truth bounding boxes."""
```

(continues on next page)

(continued from previous page)

```
    batch_size, anchors = labels.shape[0], anchors.squeeze(0)
    batch_offset, batch_mask, batch_class_labels = [], [], []
    device, num_anchors = anchors.device, anchors.shape[0]
    for i in range(batch_size):
        label = labels[i, :, :]
        anchors_bbox_map = assign_anchor_to_bbox(
            label[:, 1:], anchors, device)
        bbox_mask = ((anchors_bbox_map >= 0).float().unsqueeze(-1)).repeat(
            1, 4)
        # Initialize class labels and assigned bounding box coordinates with
        # zeros
        class_labels = torch.zeros(num_anchors, dtype=torch.long,
                                   device=device)
        assigned_bb = torch.zeros((num_anchors, 4), dtype=torch.float32,
                                  device=device)
        # Label classes of anchor boxes using their assigned ground-truth
        # bounding boxes. If an anchor box is not assigned any, we label its
        # class as background (the value remains zero)
        indices_true = torch.nonzero(anchors_bbox_map >= 0)
        bb_idx = anchors_bbox_map[indices_true]
        class_labels[indices_true] = label[bb_idx, 0].long() + 1
        assigned_bb[indices_true] = label[bb_idx, 1:]
        # Offset transformation
        offset = offset_boxes(anchors, assigned_bb) * bbox_mask
        batch_offset.append(offset.reshape(-1))
        batch_mask.append(bbox_mask.reshape(-1))
        batch_class_labels.append(class_labels)
    bbox_offset = torch.stack(batch_offset)
    bbox_mask = torch.stack(batch_mask)
    class_labels = torch.stack(batch_class_labels)
    return (bbox_offset, bbox_mask, class_labels)
```

```
#@save
def multibox_target(anchors, labels):
    """Label anchor boxes using ground-truth bounding boxes."""
    batch_size, anchors = labels.shape[0], anchors.squeeze(0)
    batch_offset, batch_mask, batch_class_labels = [], [], []
    device, num_anchors = anchors.ctx, anchors.shape[0]
    for i in range(batch_size):
        label = labels[i, :, :]
        anchors_bbox_map = assign_anchor_to_bbox(
            label[:, 1:], anchors, device)
        bbox_mask = np.tile((np.expand_dims((anchors_bbox_map >= 0),
                                            axis=-1)), (1, 4)).astype('int32')
        # Initialize class labels and assigned bounding box coordinates with
        # zeros
        class_labels = d2l.zeros(num_anchors, dtype=np.int32, ctx=device)
        assigned_bb = d2l.zeros((num_anchors, 4), dtype=np.float32,
                                ctx=device)
        # Label classes of anchor boxes using their assigned ground-truth
        # bounding boxes. If an anchor box is not assigned any, we label its
        # class as background (the value remains zero)
        indices_true = np.nonzero(anchors_bbox_map >= 0)[0]
        bb_idx = anchors_bbox_map[indices_true]
```

(continues on next page)

(continued from previous page)

```
        class_labels[indices_true] = label[bb_idx, 0].astype('int32') + 1
        assigned_bb[indices_true] = label[bb_idx, 1:]
        # Offset transformation
        offset = offset_boxes(anchors, assigned_bb) * bbox_mask
        batch_offset.append(offset.reshape(-1))
        batch_mask.append(bbox_mask.reshape(-1))
        batch_class_labels.append(class_labels)
    bbox_offset = d2l.stack(batch_offset)
    bbox_mask = d2l.stack(batch_mask)
    class_labels = d2l.stack(batch_class_labels)
    return (bbox_offset, bbox_mask, class_labels)
```

```
anchors = d2l.tensor([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                      [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = d2l.tensor([0] * d2l.size(anchors))
cls_probs = d2l.tensor([[0] * 4,  # Predicted background likelihood
                      [0.9, 0.8, 0.7, 0.1],  # Predicted dog likelihood
                      [0.1, 0.2, 0.3, 0.9]])  # Predicted cat likelihood
```

```
anchors = d2l.tensor([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                      [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = d2l.tensor([0] * d2l.size(anchors))
cls_probs = d2l.tensor([[0] * 4,  # Predicted background likelihood
                      [0.9, 0.8, 0.7, 0.1],  # Predicted dog likelihood
                      [0.1, 0.2, 0.3, 0.9]])  # Predicted cat likelihood
```

例

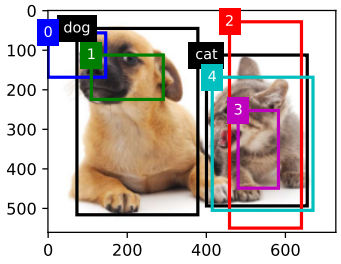
具体例を通してアンカーボックスのラベル付けを説明しよう。読み込んだ画像中の犬と猫に対する真のバウンディングボックスを定義する。ここで最初の要素はクラス（犬は 0、猫は 1）で、残りの 4 要素は左上隅と右下隅の (x, y) 軸座標（範囲は 0 から 1）である。また、左上隅と右下隅の座標を用いてラベル付けする 5 つのアンカーボックス $A_0, \ldots, A_4$（インデックスは 0 から始まる）を構成する。そして、これらの真のバウンディングボックスとアンカーボックスを画像上に描画する。

```
ground_truth = d2l.tensor([[0, 0.1, 0.08, 0.52, 0.92],
                           [1, 0.55, 0.2, 0.9, 0.88]])
anchors = d2l.tensor([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                      [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                      [0.57, 0.3, 0.92, 0.9]])

fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```

```
ground_truth = d2l.tensor([[0, 0.1, 0.08, 0.52, 0.92],
```

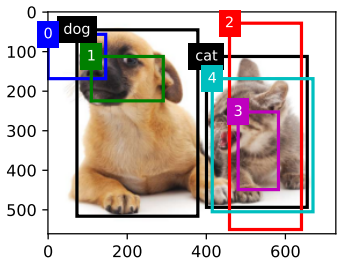
(continues on next page)



(continued from previous page)

```
                          [1, 0.55, 0.2, 0.9, 0.88]])
anchors = d2l.tensor([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                      [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                      [0.57, 0.3, 0.92, 0.9]])

fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```




```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```


```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```


上で定義した multibox_target 関数を用いると、犬と猫の真のバウンディングボックスに基づいて、これらのアンカーボックスのクラスとオフセットにラベル付けできる。この例では、背景、犬、猫のクラスのインデックスはそれぞれ0、1、2である。以下では、アンカーボックスと真のバウンディングボックスの例のために次元を追加する。

```
labels = multibox_target(anchors.unsqueeze(dim=0),
                         ground_truth.unsqueeze(dim=0))
```

```
labels = multibox_target(np.expand_dims(anchors, axis=0),
                         np.expand_dims(ground_truth, axis=0))
```

```
fig = d2l.plt.imshow(img)
for i in d2l.numpy(output[0]):
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [d2l.tensor(i[2:]) * bbox_scale], label)
```


```
fig = d2l.plt.imshow(img)
for i in d2l.numpy(output[0]):
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [d2l.tensor(i[2:]) * bbox_scale], label)
```


返される結果は 3 つの項目からなり、いずれもテンソル形式である。3 番目の項目には、入力アンカーボックスのラベル付きクラスが含まれる。

以下では、画像内のアンカーボックスと真のバウンディングボックスの位置に基づいて、返されたクラスラベルを解析しよう。まず、すべてのアンカーボックスと真のバウンディングボックスの組の中で、アンカーボックス A_4 と猫の真のバウンディングボックスの IoU が最大である。したがって、A_4 のクラスは猫としてラベル付けされる。A_4 または猫の真のバウンディングボックスを含む組を除くと、残りの中ではアンカーボックス A_1 と犬の真のバウンディングボックスの組の IoU が最大である。したがって、A_1 のクラスは犬としてラベル付けされる。次に、残りの 3 つの未ラベルのアンカーボックス A_0, A_2, A_3 を走査する必要がある。A_0 については、IoU が最大となる真のバウンディングボックスのクラスは犬であるが、IoU はあらかじめ定めたしきい値（0.5）を下回るため、クラスは背景としてラベル付けされる。A_2 については、IoU が最大となる真のバウンディングボックスのクラスは猫で、IoU はしきい値を超えるため、クラスは猫としてラベル付けされる。A_3 については、IoU が最大となる真のバウンディングボックスのクラスは猫であるが、その値はしきい値を下回るため、クラスは背景としてラベル付けされる。

```
labels[2]
```

```
tensor([[0, 1, 2, 0, 2]])
```

2 番目に返される項目は、形状 (batch size, four times the number of anchor boxes) のマスク変数である。マスク変数の 4 要素ごとに、各アンカーボックスの 4 つのオフセット値に対応する。背景検出は考慮しないので、この負例クラスのオフセットは目的関数に影響すべきではない。要素ごとの乗算を通じて、マスク変数の 0 は、目的関数を計算する前に負例クラスのオフセットを除外する。

```
labels[1]
```

```
tensor([[0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 1., 1.
↪,
        1., 1.]])
```

1 番目に返される項目には、各アンカーボックスに対してラベル付けされた 4 つのオフセット値が含まれる。負例クラスのアンカーボックスのオフセットは 0 としてラベル付けされることに注意しよ。

```
labels[0]
```

```
tensor([[-0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00,  1.40e+00,  1.00e+01,
          2.59e+00,  7.18e+00, -1.20e+00,  2.69e-01,  1.68e+00, -1.57e+00,
         -0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00, -5.71e-01, -1.00e+00,
          4.17e-06,  6.26e-01]])
```

17.4.4 非最大抑制によるバウンディングボックスの予測

予測時には、画像に対して複数のアンカーボックスを生成し、それぞれについてクラスとオフセットを予測する。したがって、アンカーボックスとその予測オフセットに基づいて 予測バウンディングボックスが得られる。以下では、アンカーボックスとオフセット予測を入力として受け取り、逆オフセット変換を適用して予測バウンディングボックスの座標を返す `offset_inverse` 関数を実装する。

```
#@save
def offset_inverse(anchors, offset_preds):
    """Predict bounding boxes based on anchor boxes with predicted offsets."""
    anc = d2l.box_corner_to_center(anchors)
    pred_bbox_xy = (offset_preds[:, :2] * anc[:, 2:] / 10) + anc[:, :2]
    pred_bbox_wh = d2l.exp(offset_preds[:, 2:] / 5) * anc[:, 2:]
    pred_bbox = d2l.concat((pred_bbox_xy, pred_bbox_wh), axis=1)
    predicted_bbox = d2l.box_center_to_corner(pred_bbox)
    return predicted_bbox
```

アンカーボックスが多数あると、同じ物体を囲む非常に似た（大きく重なった）予測バウンディングボックスが多数出力される可能性がある。出力を簡潔にするために、非最大抑制（NMS）を用いて、同じ物体に属する似た予測バウンディングボックスをまとめることができる。

非最大抑制の仕組みは次のとおりである。予測バウンディングボックス B について、物体検出モデルは各クラスの予測確率を計算する。最大の予測確率を p とすると、この確率に対応するクラスが B の予測クラスである。特に、この確率 p を予測バウンディングボックス B の *confidence*（スコア）と呼ぶ。同じ画像上で、背景以外のすべての予測バウンディングボックスを confidence の降順

に並べてリスト L を作りる。その後、並べ替えたリスト L を次の手順で操作する。

1. 最も confidence の高い予測バウンディングボックス B_1 を基準として選び、B_1 との IoU があらかじめ定めたしきい値 ϵ を超える、基準以外の予測バウンディングボックスをすべて L から取り除きる。この時点で、L は最も confidence の高い予測バウンディングボックスを保持し、それに似すぎている他のものを削除する。要するに、最大でない confidence スコアをもつものは 抑制される。

2. 2 番目に confidence の高い予測バウンディングボックス B_2 を別の基準として選び、B_2 との IoU が ϵ を超える、基準以外の予測バウンディングボックスをすべて L から取り除きる。

3. リスト L のすべての予測バウンディングボックスが基準として使われるまで、上記の処理を繰り返す。この時点で、L に含まれる任意の 2 つの予測バウンディングボックスの IoU はしきい値 ϵ 未満である。したがって、どの 2 つも互いに似すぎていない。

4. リスト L に含まれるすべての予測バウンディングボックスを出力する。

次の nms 関数は confidence スコアを降順に並べ、そのインデックスを返す。

```
#@save
def nms(boxes, scores, iou_threshold):
    """Sort confidence scores of predicted bounding boxes."""
    B = torch.argsort(scores, dim=-1, descending=True)
    keep = []  # Indices of predicted bounding boxes that will be kept
    while B.numel() > 0:
        i = B[0]
        keep.append(i)
        if B.numel() == 1: break
        iou = box_iou(boxes[i, :].reshape(-1, 4),
                      boxes[B[1:], :].reshape(-1, 4)).reshape(-1)
        inds = torch.nonzero(iou <= iou_threshold).reshape(-1)
        B = B[inds + 1]
    return d2l.tensor(keep, device=boxes.device)
```

```
#@save
def nms(boxes, scores, iou_threshold):
    """Sort confidence scores of predicted bounding boxes."""
    B = scores.argsort()[::-1]
    keep = []  # Indices of predicted bounding boxes that will be kept
    while B.size > 0:
        i = B[0]
        keep.append(i)
        if B.size == 1: break
        iou = box_iou(boxes[i, :].reshape(-1, 4),
                      boxes[B[1:], :].reshape(-1, 4)).reshape(-1)
        inds = np.nonzero(iou <= iou_threshold)[0]
        B = B[inds + 1]
    return np.array(keep, dtype=np.int32, ctx=boxes.ctx)
```

以下の multibox_detection を定義して、予測バウンディングボックスに非最大抑制を適用する。実装が少し複雑に見えても心配はいらない。実装の直後に、具体例を用いて動作を示す。

```
#@save
def multibox_detection(cls_probs, offset_preds, anchors, nms_threshold=0.5,
                       pos_threshold=0.009999999):
    """Predict bounding boxes using non-maximum suppression."""
    device, batch_size = cls_probs.device, cls_probs.shape[0]
    anchors = anchors.squeeze(0)
    num_classes, num_anchors = cls_probs.shape[1], cls_probs.shape[2]
    out = []
    for i in range(batch_size):
        cls_prob, offset_pred = cls_probs[i], offset_preds[i].reshape(-1, 4)
        conf, class_id = torch.max(cls_prob[1:], 0)
        predicted_bb = offset_inverse(anchors, offset_pred)
        keep = nms(predicted_bb, conf, nms_threshold)
        # Find all non-`keep` indices and set the class to background
        all_idx = torch.arange(num_anchors, dtype=torch.long, device=device)
        combined = torch.cat((keep, all_idx))
        uniques, counts = combined.unique(return_counts=True)
        non_keep = uniques[counts == 1]
        all_id_sorted = torch.cat((keep, non_keep))
        class_id[non_keep] = -1
        class_id = class_id[all_id_sorted]
        conf, predicted_bb = conf[all_id_sorted], predicted_bb[all_id_sorted]
        # Here `pos_threshold` is a threshold for positive (non-background)
        # predictions
        below_min_idx = (conf < pos_threshold)
        class_id[below_min_idx] = -1
        conf[below_min_idx] = 1 - conf[below_min_idx]
        pred_info = torch.cat((class_id.unsqueeze(1),
                               conf.unsqueeze(1),
                               predicted_bb), dim=1)
        out.append(pred_info)
    return d2l.stack(out)
```

```
#@save
def multibox_detection(cls_probs, offset_preds, anchors, nms_threshold=0.5,
                       pos_threshold=0.009999999):
    """Predict bounding boxes using non-maximum suppression."""
    device, batch_size = cls_probs.ctx, cls_probs.shape[0]
    anchors = np.squeeze(anchors, axis=0)
    num_classes, num_anchors = cls_probs.shape[1], cls_probs.shape[2]
    out = []
    for i in range(batch_size):
        cls_prob, offset_pred = cls_probs[i], offset_preds[i].reshape(-1, 4)
        conf, class_id = np.max(cls_prob[1:], 0), np.argmax(cls_prob[1:], 0)
        predicted_bb = offset_inverse(anchors, offset_pred)
        keep = nms(predicted_bb, conf, nms_threshold)
        # Find all non-`keep` indices and set the class to background
        all_idx = np.arange(num_anchors, dtype=np.int32, ctx=device)
        combined = d2l.concat((keep, all_idx))
        unique, counts = np.unique(combined, return_counts=True)
        non_keep = unique[counts == 1]
```

(continues on next page)

(continued from previous page)

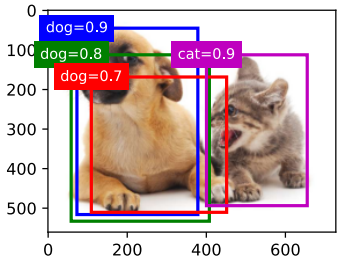
```
        all_id_sorted = d2l.concat((keep, non_keep))
        class_id[non_keep] = -1
        class_id = class_id[all_id_sorted].astype('float32')
        conf, predicted_bb = conf[all_id_sorted], predicted_bb[all_id_sorted]
        # Here `pos_threshold` is a threshold for positive (non-background)
        # predictions
        below_min_idx = (conf < pos_threshold)
        class_id[below_min_idx] = -1
        conf[below_min_idx] = 1 - conf[below_min_idx]
        pred_info = d2l.concat((np.expand_dims(class_id, axis=1),
                                np.expand_dims(conf, axis=1),
                                predicted_bb), axis=1)
        out.append(pred_info)
    return d2l.stack(out)
```

それでは、4 つのアンカーボックスを用いた具体例に上記の実装を適用してみよう。簡単のため、予測オフセットはすべて 0 と仮定する。これは、予測バウンディングボックスがアンカーボックスと同じであることを意味する。背景、犬、猫の各クラスについて、それぞれの予測確率も定義する。

```
anchors = d2l.tensor([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                      [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = d2l.tensor([0] * d2l.size(anchors))
cls_probs = d2l.tensor([[0] * 4,  # Predicted background likelihood
                        [0.9, 0.8, 0.7, 0.1],  # Predicted dog likelihood
                        [0.1, 0.2, 0.3, 0.9]])  # Predicted cat likelihood
```

これらの予測バウンディングボックスとその confidence を画像上に描画できる。

```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```



ここで `multibox_detection` 関数を呼び出して、しきい値を 0.5 に設定した非最大抑制を実行できる。なお、テンソル入力には例の次元を追加している。

返される結果の形状は (batch size, number of anchor boxes, 6) である。最内側の次

元の 6 要素は、同じ予測バウンディングボックスに対する出力情報を与える。1 番目の要素は予測クラスのインデックスで、0 から始まります（0 は犬、1 は猫です）。値 -1 は背景または非最大抑制による除去を表す。2 番目の要素は予測バウンディングボックスの confidence である。残りの 4 要素は、それぞれ予測バウンディングボックスの左上隅と右下隅の (x, y) 軸座標（範囲は 0 から 1）である。

```
output = multibox_detection(cls_probs.unsqueeze(dim=0),
                            offset_preds.unsqueeze(dim=0),
                            anchors.unsqueeze(dim=0),
                            nms_threshold=0.5)
output
```

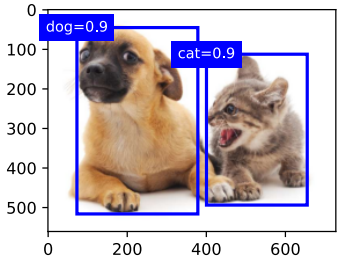
```
tensor([[[ 0.00,  0.90,  0.10,  0.08,  0.52,  0.92],
         [ 1.00,  0.90,  0.55,  0.20,  0.90,  0.88],
         [-1.00,  0.80,  0.08,  0.20,  0.56,  0.95],
         [-1.00,  0.70,  0.15,  0.30,  0.62,  0.91]]])
```

```
output = multibox_detection(np.expand_dims(cls_probs, axis=0),
                            np.expand_dims(offset_preds, axis=0),
                            np.expand_dims(anchors, axis=0),
                            nms_threshold=0.5)
output
```

```
array([[[ 1.  ,  0.9 ,  0.55,  0.2 ,  0.9 ,  0.88],
        [ 0.  ,  0.9 ,  0.1 ,  0.08,  0.52,  0.92],
        [-1.  ,  0.8 ,  0.08,  0.2 ,  0.56,  0.95],
        [-1.  ,  0.7 ,  0.15,  0.3 ,  0.62,  0.91]]])
```

クラス -1 の予測バウンディングボックスを取り除いた後、非最大抑制によって保持された最終的な予測バウンディングボックスを出力できる。

```
fig = d2l.plt.imshow(img)
for i in d2l.numpy(output[0]):
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [d2l.tensor(i[2:]) * bbox_scale], label)
```



実際には、非最大抑制を行う前に、confidence の低い予測バウンディングボックスを取り除くことで、このアルゴリズムの計算量を減らせる。また、非最大抑制の出力に後処理を施し、たとえば最終出力ではより高い confidence をもつ結果だけを保持することもできる。

17.4.5 まとめ

- 画像の各ピクセルを中心として、異なる形状のアンカーボックスを生成する。
- Intersection over union（IoU）は Jaccard index とも呼ばれ、2 つのバウンディングボックスの類似度を測る。これは共通部分の面積を和集合の面積で割った比である。
- 学習データセットでは、各アンカーボックスに対して 2 種類のラベルが必要である。1 つはアンカーボックスに対応する物体のクラス、もう 1 つはアンカーボックスに対する真のバウンディングボックスのオフセットである。
- 予測時には、非最大抑制（NMS）を用いて似た予測バウンディングボックスを取り除き、出力を簡潔にできる。

17.4.6 演習

1. `multibox_prior` 関数の `sizes` と `ratios` の値を変更してみよ。生成されるアンカーボックスはどう変わりますか。
2. IoU が 0.5 となる 2 つのバウンディングボックスを構成し、可視化しよ。それらはどのように重なっているか。
3. Section 17.4.3 と Section 17.4.4 の `anchors` 変数を変更しよ。結果はどう変わりますか。
4. 非最大抑制は、予測バウンディングボックスを 取り除くことで抑制する貪欲法である。取り除かれたものの中に、実は有用なものが含まれている可能性はあるか。ソフトに抑制するようにこのアルゴリズムをどのように修正できるか。Soft-NMS (Bodla *et al.*, 2017) を参照してもよいである。
5. 手作業で設計するのではなく、非最大抑制を学習することは可能だろうか。

17.5 マルチスケール物体検出

Section 17.4 では、入力画像の各画素を中心とする複数のアンカーボックスを生成した。
本質的には、これらのアンカーボックスは画像のさまざまな領域のサンプルを表している。

しかし、すべての画素に対して生成すると、計算すべきアンカーボックスが多すぎる場合がある。
561 × 728 の入力画像を考えてみよう。
各画素を中心として、形状の異なる 5 個のアンカーボックスを生成すると、200 万個を超えるアンカーボックス（561 × 728 × 5）に対して、画像上でラベル付けと予測を行う必要がある。

17.5.1 マルチスケールのアンカーボックス

画像上のアンカーボックスを減らすのは、それほど難しくないと気づくかもしれない。
たとえば、入力画像から画素を少数だけ一様にサンプリングし、その画素を中心とするアンカーボックスを生成すればよいのである。
さらに、異なるスケールで、異なるサイズのアンカーボックスを異なる個数だけ生成できる。
直感的には、小さな物体のほうが大きな物体よりも画像中に現れやすいである。
例として、2 × 2 の画像には、1 × 1、1 × 2、2 × 2 の物体がそれぞれ 4 通り、2 通り、1 通りの現れ方で存在しえる。
したがって、小さな物体を検出するために小さなアンカーボックスを使う場合は、より多くの領域をサンプリングでき、
一方で大きな物体に対しては、より少ない領域をサンプリングすればよいのである。

複数のスケールでアンカーボックスを生成する方法を示すために、画像を読み込みよう。
その高さと幅はそれぞれ 561 ピクセルと 728 ピクセルである。

```
%matplotlib inline
from d2l import torch as d2l
import torch

img = d2l.plt.imread('../img/catdog.jpg')
h, w = img.shape[:2]
h, w
```

```
(561, 728)
```

```
%matplotlib inline
from d2l import mxnet as d2l
```

(continues on next page)

(continued from previous page)

```
from mxnet import image, np, npx

npx.set_np()

img = image.imread('../img/catdog.jpg')
h, w = img.shape[:2]
h, w
```

```
[06:59:11] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(561, 728)
```

```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```


```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```


Section 10.2 で、畳み込み層の 2 次元配列出力を特徴マップと呼ぶことを思い出してほしい。
特徴マップの形状を定義することで、任意の画像上で一様にサンプリングされたアンカーボックスの中心を決定できる。

以下に display_anchors 関数を定義する。
特徴マップ (fmap) 上の各単位（画素）をアンカーボックスの中心としてアンカーボックス (anchors) を生成する。
アンカーボックス (anchors) の (x, y) 軸の座標値は、特徴マップ (fmap) の幅と高さで割られているため、
これらの値は 0 から 1 の間にあり、特徴マップ上でのアンカーボックスの相対位置を表す。

アンカーボックス (anchors) の中心は特徴マップ (fmap) 上のすべての単位に広がっているので、
これらの中心は、相対的な空間位置の観点から、任意の入力画像上で一様に分布していなければならない。
より具体的には、特徴マップ fmap_w と fmap_h の幅と高さがそれぞれ与えられたとき、
以下の関数は任意の入力画像上で fmap_h 行 fmap_w 列の画素を一様にサンプリングする。

この一様にサンプリングされた画素を中心として、スケール s のアンカーボックス（s のリスト長が 1 であると仮定）と、異なるアスペクト比（ratios）を持つアンカーボックスが生成される。

```
def display_anchors(fmap_w, fmap_h, s):
    d2l.set_figsize()
    # Values on the first two dimensions do not affect the output
    fmap = d2l.zeros((1, 10, fmap_h, fmap_w))
    anchors = d2l.multibox_prior(fmap, sizes=s, ratios=[1, 2, 0.5])
    bbox_scale = d2l.tensor((w, h, w, h))
    d2l.show_bboxes(d2l.plt.imshow(img).axes,
                    anchors[0] * bbox_scale)
```

```
def display_anchors(fmap_w, fmap_h, s):
    d2l.set_figsize()
    # Values on the first two dimensions do not affect the output
    fmap = np.zeros((1, 10, fmap_h, fmap_w))
    anchors = npx.multibox_prior(fmap, sizes=s, ratios=[1, 2, 0.5])
    bbox_scale = np.array((w, h, w, h))
    d2l.show_bboxes(d2l.plt.imshow(img.asnumpy()).axes,
                    anchors[0] * bbox_scale)
```

```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```

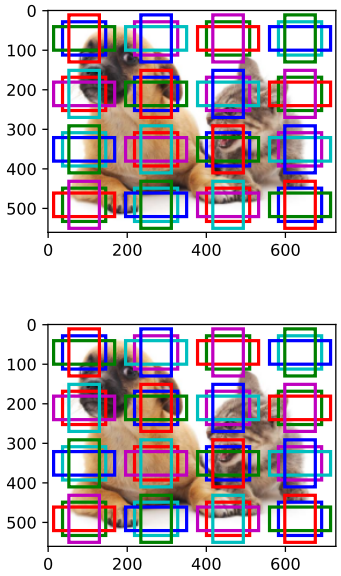

```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```


まず、小さな物体の検出を考えよう。
表示時に見分けやすくするため、ここでは中心の異なるアンカーボックス同士が重ならないようにしている。
アンカーボックスのスケールは 0.15 に設定し、特徴マップの高さと幅は 4 に設定する。
画像上の 4 行 4 列にあるアンカーボックスの中心が一様に分布していることがわかる。

```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```


```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```

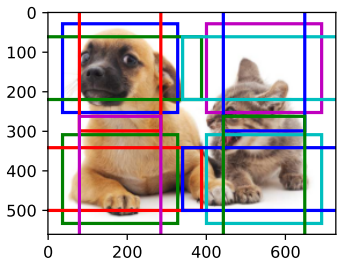

```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```




```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```

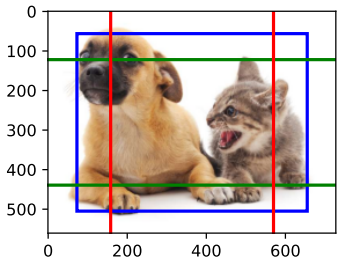
次に、特徴マップの高さと幅を半分に減らし、より大きなアンカーボックスを使って大きな物体を検出することを考える。
スケールを 0.4 に設定すると、いくつかのアンカーボックスは互いに重なる。

```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```



最後に、特徴マップの高さと幅をさらに半分に減らし、アンカーボックスのスケールを 0.8 に増やすと、
アンカーボックスの中心は画像の中心になる。

```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```



17.5.2 マルチスケール検出

マルチスケールのアンカーボックスを生成したので、
それらを用いてさまざまなサイズの物体を異なるスケールで検出す。
以下では、CNN ベースのマルチスケール物体検出手法を紹介する。これは
Section 17.7 で実装する。

あるスケールで、たとえば形状が $h \times w$ の c 個の特徴マップがあるとする。
Section 17.5.1 の方法を用いると、hw 組のアンカーボックスを生成できる。
各組は同じ中心を持つ a 個のアンカーボックスからなる。
たとえば、Section 17.5.1 の実験の最初のスケールでは、
10 個（チャネル数）の 4×4 特徴マップが与えられたとき、
16 組のアンカーボックスを生成した。各組には同じ中心を持つ 3 個のアンカーボックスが含まれる。
次に、各アンカーボックスに対して、正解のバウンディングボックスに基づいてクラスとオフセットをラベル付けする。
現在のスケールでは、物体検出モデルは入力画像上の hw 組のアンカーボックスのクラスとオフセットを予測する必要があり、
異なる組は異なる中心を持つ。

ここでの c 個の特徴マップは、入力画像に基づく CNN の順伝播によって得られた中間出力だと仮定する。
各特徴マップには hw 個の異なる空間位置があるので、同じ空間位置は c 個の単位を持つと考えられる。
Section 10.2 における受容野の定義によれば、
特徴マップの同じ空間位置にあるこれら c 個の単位は、入力画像上で同じ受容野を持つ。
つまり、それらは同じ受容野における入力画像情報を表している。

したがって、特徴マップの同じ空間位置にある c 個の単位を、
この空間位置を用いて生成された a 個のアンカーボックスのクラスとオフセットへ変換できる。
本質的には、ある受容野における入力画像の情報を用いて、
その受容野に近い入力画像上のアンカーボックスのクラスとオフセットを予測しているのである。

異なる層の特徴マップが、入力画像上で異なる大きさの受容野を持つ場合、それらは異なるサイズの物体の検出に利用できる。
たとえば、出力層に近い特徴マップの単位ほど広い受容野を持つようなニューラルネットワークを設計すれば、
入力画像からより大きな物体を検出できる。

要するに、深層ニューラルネットワークによる画像の層ごとの表現を複数のレベルで活用することで、
マルチスケール物体検出を実現できる。
これがどのように機能するかは、Section 17.7 の具体例で示す。

17.5.3 要約

- 複数のスケールで、異なるサイズの物体を検出するために、異なるサイズのアンカーボックスを生成できる。
- 特徴マップの形状を定義することで、任意の画像上で一様にサンプリングされたアンカーボックスの中心を決定できる。
- ある受容野における入力画像の情報を用いて、その受容野に近い入力画像上のアンカーボックスのクラスとオフセットを予測する。
- 深層学習により、画像の層ごとの表現を複数のレベルで活用して、マルチスケール物体検出を行える。

17.5.4 演習

1. Section 11.1 での議論によれば、深層ニューラルネットワークは、画像に対して抽象度が高まる階層的な特徴を学習する。マルチスケール物体検出では、異なるスケールの特徴マップは異なる抽象度のレベルに対応するのだろうか。なぜそう言えるのか、あるいは言えないのか。
2. Section 17.5.1 の実験の最初のスケール（`fmap_w=4, fmap_h=4`）で、重なりうる一様分布のアンカーボックスを生成しなさい。
3. 形状が $1 \times c \times h \times w$ の特徴マップ変数が与えられたとする。ここで、c, h, w はそれぞれ特徴マップのチャネル数、高さ、幅である。この変数をアンカーボ

ックスのクラスとオフセットにどのように変換できるか。出力の形状はどうなるか。

17.6 物体検出データセット

物体検出の分野には、MNIST や Fashion-MNIST のような小規模データセットはない。物体検出モデルを手早く示すために、私たちは小さなデータセットを収集してラベル付けした。まず、オフィスにあった自由に使えるバナナの写真を撮り、回転やサイズの異なる 1000 枚のバナナ画像を生成した。次に、それぞれのバナナ画像をいくつかの背景画像上のランダムな位置に配置した。最後に、それらの画像中のバナナに対してバウンディングボックスをラベル付けした。

17.6.1 データセットのダウンロード

画像ファイルと csv ラベルファイルをすべて含むバナナ検出データセットは、インターネットから直接ダウンロードできる。

```
%matplotlib inline
from d2l import torch as d2l
import torch
import torchvision
import os
import pandas as pd
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, np, npx
import os
import pandas as pd

npx.set_np()
```

```
#@save
d2l.DATA_HUB['banana-detection'] = (
    d2l.DATA_URL + 'banana-detection.zip',
    '5de26c8fce5ccdea9f91267273464dc968d20d72')
```

```
#@save
d2l.DATA_HUB['banana-detection'] = (
    d2l.DATA_URL + 'banana-detection.zip',
    '5de26c8fce5ccdea9f91267273464dc968d20d72')
```

```
#@save
d2l.DATA_HUB['banana-detection'] = (
    d2l.DATA_URL + 'banana-detection.zip',
    '5de26c8fce5ccdea9f91267273464dc968d20d72')
```


```
#@save
d2l.DATA_HUB['banana-detection'] = (
    d2l.DATA_URL + 'banana-detection.zip',
    '5de26c8fce5ccdea9f91267273464dc968d20d72')
```


```
batch_size, edge_size = 32, 256
train_iter, _ = load_data_bananas(batch_size)
batch = next(iter(train_iter))
batch[0].shape, batch[1].shape
```


```
batch_size, edge_size = 32, 256
train_iter, _ = load_data_bananas(batch_size)
batch = next(iter(train_iter))
batch[0].shape, batch[1].shape
```


17.6.2 データセットの読み込み

以下の read_data_bananas 関数でバナナ検出データセットを読み込みます。このデータセットには、物体クラスのラベルと左上および右下の角の正解バウンディングボックス座標を含む csv ファイルがある。

```
#@save
def read_data_bananas(is_train=True):
    """Read the banana detection dataset images and labels."""
    data_dir = d2l.download_extract('banana-detection')
    csv_fname = os.path.join(data_dir, 'bananas_train' if is_train
                             else 'bananas_val', 'label.csv')
    csv_data = pd.read_csv(csv_fname)
    csv_data = csv_data.set_index('img_name')
    images, targets = [], []
    for img_name, target in csv_data.iterrows():
        images.append(torchvision.io.read_image(
            os.path.join(data_dir, 'bananas_train' if is_train else
                         'bananas_val', 'images', f'{img_name}')))
        # Here `target` contains (class, upper-left x, upper-left y,
        # lower-right x, lower-right y), where all the images have the same
        # banana class (index 0)
        targets.append(list(target))
    return images, torch.tensor(targets).unsqueeze(1) / 256
```

```
#@save
def read_data_bananas(is_train=True):
    """Read the banana detection dataset images and labels."""
```

(continues on next page)

(continued from previous page)

```
    data_dir = d2l.download_extract('banana-detection')
    csv_fname = os.path.join(data_dir, 'bananas_train' if is_train
                             else 'bananas_val', 'label.csv')
    csv_data = pd.read_csv(csv_fname)
    csv_data = csv_data.set_index('img_name')
    images, targets = [], []
    for img_name, target in csv_data.iterrows():
        images.append(image.imread(
            os.path.join(data_dir, 'bananas_train' if is_train else
                         'bananas_val', 'images', f'{img_name}')))
        # Here `target` contains (class, upper-left x, upper-left y,
        # lower-right x, lower-right y), where all the images have the same
        # banana class (index 0)
        targets.append(list(target))
    return images, np.expand_dims(np.array(targets), 1) / 256
```

read_data_bananas 関数を使って画像とラベルを読み込むことで、以下の BananasDataset クラスを用いて、バナナ検出データセットを読み込むためのカスタマイズした Dataset インスタンスを作成できる。

```
#@save
class BananasDataset(torch.utils.data.Dataset):
    """A customized dataset to load the banana detection dataset."""
    def __init__(self, is_train):
        self.features, self.labels = read_data_bananas(is_train)
        print('read ' + str(len(self.features)) + (f' training examples' if
              is_train else f' validation examples'))

    def __getitem__(self, idx):
        return (self.features[idx].float(), self.labels[idx])

    def __len__(self):
        return len(self.features)
```

```
#@save
class BananasDataset(gluon.data.Dataset):
    """A customized dataset to load the banana detection dataset."""
    def __init__(self, is_train):
        self.features, self.labels = read_data_bananas(is_train)
        print('read ' + str(len(self.features)) + (f' training examples' if
              is_train else f' validation examples'))

    def __getitem__(self, idx):
        return (self.features[idx].astype('float32').transpose(2, 0, 1),
                self.labels[idx])

    def __len__(self):
        return len(self.features)
```

最後に、学習用とテスト用の両方について 2 つのデータ反復子インスタンスを返す load_data_bananas 関数を定義する。テストデータセットについては、ランダムな順序で読み込む必要はない。

```
#@save
def load_data_bananas(batch_size):
    """Load the banana detection dataset."""
    train_iter = torch.utils.data.DataLoader(BananasDataset(is_train=True),
                                             batch_size, shuffle=True)
    val_iter = torch.utils.data.DataLoader(BananasDataset(is_train=False),
                                           batch_size)
    return train_iter, val_iter
```

```
#@save
def load_data_bananas(batch_size):
    """Load the banana detection dataset."""
    train_iter = gluon.data.DataLoader(BananasDataset(is_train=True),
                                       batch_size, shuffle=True)
    val_iter = gluon.data.DataLoader(BananasDataset(is_train=False),
                                     batch_size)
    return train_iter, val_iter
```

ミニバッチを 1 つ読み込み、このミニバッチ内の画像とラベルの両方の形状を表示してみましょう。画像ミニバッチの形状（バッチサイズ、チャネル数、高さ、幅）は、見覚えがあるはずである。これは、以前の画像分類タスクと同じである。ラベルミニバッチの形状は（バッチサイズ、m、5）であり、ここで m はデータセット中のどの画像が持つバウンディングボックス数としてもあり得る最大値である。

ミニバッチでの計算はより効率的であるが、連結によってミニバッチを作るには、すべての画像例が同じ数のバウンディングボックスを含んでいる必要がある。一般には、画像ごとにバウンディングボックスの数は異なり得る。そのため、m 個未満のバウンディングボックスしか持たない画像には、m に達するまで不正なバウンディングボックスがパディングされる。その後、各バウンディングボックスのラベルは長さ 5 の配列で表される。配列の最初の要素はバウンディングボックス内の物体のクラスであり、-1 はパディング用の不正なバウンディングボックスを示す。配列の残り 4 要素は、バウンディングボックスの左上隅と右下隅の (x, y) 座標値（範囲は 0 から 1）である。バナナデータセットでは、各画像に 1 つのバウンディングボックスしかないため、$m = 1$ である。

```
batch_size, edge_size = 32, 256
train_iter, _ = load_data_bananas(batch_size)
batch = next(iter(train_iter))
batch[0].shape, batch[1].shape
```

```
Downloading ../data/banana-detection.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/banana-detection.zip...
read 1000 training examples
read 100 validation examples
```

```
(torch.Size([32, 3, 256, 256]), torch.Size([32, 1, 5]))
```

17.6.3 デモンストレーション

ラベル付けされた正解バウンディングボックス付きの 10 枚の画像を示しよう。これらの画像全体で、バナナの回転、サイズ、位置が異なっていることがわかる。もちろん、これは単純な人工データセットにすぎない。実際には、現実世界のデータセットは通常もっと複雑である。

```
imgs = (batch[0][:10].permute(0, 2, 3, 1)) / 255
axes = d2l.show_images(imgs, 2, 5, scale=2)
for ax, label in zip(axes, batch[1][:10]):
    d2l.show_bboxes(ax, [label[0][1:5] * edge_size], colors=['w'])
```



```
imgs = (batch[0][:10].transpose(0, 2, 3, 1)) / 255
axes = d2l.show_images(imgs, 2, 5, scale=2)
for ax, label in zip(axes, batch[1][:10]):
    d2l.show_bboxes(ax, [label[0][1:5] * edge_size], colors=['w'])
```



17.6.4 まとめ

- 私たちが収集したバナナ検出データセットは、物体検出モデルのデモに利用できる。

- 物体検出のデータ読み込みは画像分類の場合と似ている。ただし、物体検出ではラベルに正解バウンディングボックスの情報も含まれるが、画像分類にはそれがない。

17.6.5 演習

1. バナナ検出データセットに含まれる他の画像についても、正解バウンディングボックスを示してみよう。バウンディングボックスと物体の観点で、どのような違いがあるか？
2. 物体検出に対して、ランダムクロップのようなデータ拡張を適用したいとする。画像分類の場合と比べて、どのように異なるか？ヒント: クロップした画像に物体の一部しか含まれない場合はどうなるだろうか。

17.7 シングルショット・マルチボックス検出

Section 17.3–sec_object-detection-dataset では、バウンディングボックス、アンカーボックス、マルチスケール物体検出、そして物体検出用データセットを紹介した。ここでは、こうした背景知識を用いて物体検出モデルであるシングルショット・マルチボックス検出（SSD）(Liu *et al.*, 2016) を設計する準備が整った。このモデルはシンプルで高速であり、広く使われている。これは数多くある物体検出モデルのうちの一つにすぎないが、この節で扱う設計原理や実装の詳細の一部は、他のモデルにも適用できる。

17.7.1 モデル

Fig. 17.7.1 はシングルショット・マルチボックス検出の設計概要を示している。このモデルは主にベースネットワークと、それに続くいくつかのマルチスケール特徴マップブロックから構成される。ベースネットワークは入力画像から特徴を抽出するためのもので、深い CNN を使うことができる。たとえば、元のシングルショット・マルチボックス検出の論文では、分類層の手前で切り詰めた VGG ネットワークを採用しており (Liu *et al.*, 2016)、ResNet も一般的に使われている。この設計により、ベースネットワークの出力する特徴マップをより大きくして、より多くのアンカーボックスを生成し、より小さな物体を検出できるようにする。その後、各マルチスケール特徴マップブロックは前のブロックからの特徴マップの高さと幅を（たとえば半分に）縮小し、特徴マップの各ユニットが入力画像上で持つ受容野を拡大できるようにする。

Section 17.5 で深層ニューラルネットワークによる画像の層ごとの表現を通じたマルチスケール物体検出の設計を思い出しよ。Fig. 17.7.1 の上部に近いマルチスケール特徴マップほど小さいであるが、受容野は大きいため、数は少ないがより大きな物体の検出に適している。

要するに、ベースネットワークといくつかのマルチスケール特徴マップブロッ

クを通じて、シングルショット・マルチボックス検出は異なるサイズを持つさまざまな数のアンカーボックスを生成し、これらのアンカーボックス（したがってバウンディングボックス）のクラスとオフセットを予測することでさまざまなサイズの物体を検出す。したがって、これはマルチスケール物体検出モデルである。

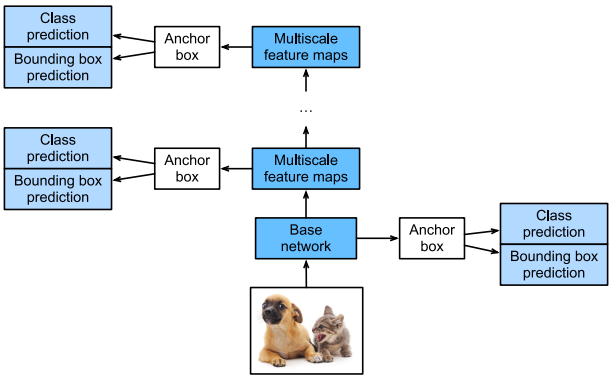


Fig. 17.7.1 マルチスケール物体検出モデルとして、シングルショット・マルチボックス検出は主にベースネットワークと、それに続くいくつかのマルチスケール特徴マップブロックから構成される。

以下では、Fig. 17.7.1 における異なるブロックの実装詳細を説明する。まず、クラス予測とバウンディングボックス予測をどのように実装するかを議論する。

クラス予測層

物体クラス数を q とする。するとアンカーボックスは $q+1$ 個のクラスを持ち、クラス 0 は背景である。あるスケールで、特徴マップの高さと幅がそれぞれ h と w であるとする。これらの特徴マップの各空間位置を中心として a 個のアンカーボックスが生成されるとき、合計 hwa 個のアンカーボックスを分類する必要がある。これは、パラメータ数が非常に多くなりがちなため、全結合層による分類を非現実的にする。Section 11.3 で畳み込み層のチャネルを使ってクラスを予測した方法を思い出しよ。シングルショット・マルチボックス検出はモデルの複雑さを下げるために同じ手法を使う。

具体的には、クラス予測層は特徴マップの幅と高さを変えずに畳み込み層を使う。このようにして、特徴マップの同じ空間次元（幅と高さ）において出力と入力の間に 1 対 1 の対応を持たせることができる。より具体的には、任意の空間位置 (x, y) における出力特徴マップのチャネルは、入力特徴マップの (x, y) を中心とするすべてのアンカーボックスに対するクラス予測を表す。有効な予測を得るには、$a(q+1)$ 個の出力チャネルが必要である。ここで、同じ空間位置に対してインデックス $i(q+1)+j$ の出力チャネルは、アンカーボックス i $(0 \le i < a)$ に対するクラス j $(0 \le j \le q)$ の予測を表す。

以下では、このようなクラス予測層を定義する。引数 `num_anchors` と `num_classes`

により、それぞれ a と q を指定する。この層は、パディング 1 の 3×3 畳み込み層を使う。この畳み込み層の入力と出力の幅と高さは変わらない。

```
%matplotlib inline
from d2l import torch as d2l
import torch
import torchvision
from torch import nn
from torch.nn import functional as F

def cls_predictor(num_inputs, num_anchors, num_classes):
    return nn.Conv2d(num_inputs, num_anchors * (num_classes + 1),
                     kernel_size=3, padding=1)
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()

def cls_predictor(num_anchors, num_classes):
    return nn.Conv2D(num_anchors * (num_classes + 1), kernel_size=3,
                     padding=1)
```

```
concat_preds([Y1, Y2]).shape
```

```
concat_preds([Y1, Y2]).shape
```

バウンディングボックス予測層

バウンディングボックス予測層の設計は、クラス予測層と似ている。唯一の違いは、各アンカーボックスに対する出力数である。ここでは $q+1$ 個のクラスではなく、4 つのオフセットを予測する必要がある。

```
def bbox_predictor(num_inputs, num_anchors):
    return nn.Conv2d(num_inputs, num_anchors * 4, kernel_size=3, padding=1)
```

```
def bbox_predictor(num_anchors):
    return nn.Conv2D(num_anchors * 4, kernel_size=3, padding=1)
```

```
sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
         [0.88, 0.961]]
ratios = [[1, 2, 0.5]] * 5
num_anchors = len(sizes[0]) + len(ratios[0]) - 1
```

```
sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
         [0.88, 0.961]]
ratios = [[1, 2, 0.5]] * 5
num_anchors = len(sizes[0]) + len(ratios[0]) - 1
```

複数スケールの予測を連結する

前述したように、シングルショット・マルチボックス検出はマルチスケール特徴マップを使ってアンカーボックスを生成し、そのクラスとオフセットを予測する。異なるスケールでは、特徴マップの形状や同じユニットを中心とするアンカーボックスの数が異なる場合がある。したがって、異なるスケールでの予測出力の形状も異なりえる。

以下の例では、同じミニバッチに対して 2 つの異なるスケールの特徴マップ `Y1` と `Y2` を構成する。ここで `Y2` の高さと幅は `Y1` の半分である。クラス予測を例に取りよう。`Y1` と `Y2` の各ユニットに対してそれぞれ 5 個と 3 個のアンカーボックスが生成されるとする。さらに、物体クラス数が 10 であるとする。特徴マップ `Y1` と `Y2` に対するクラス予測出力のチャネル数はそれぞれ $5 \times (10 + 1) = 55$ と $3 \times (10 + 1) = 33$ であり、いずれの出力形状も（バッチサイズ，チャネル数，高さ，幅）である。

```
def forward(x, block):
    return block(x)

Y1 = forward(torch.zeros((2, 8, 20, 20)), cls_predictor(8, 5, 10))
Y2 = forward(torch.zeros((2, 16, 10, 10)), cls_predictor(16, 3, 10))
Y1.shape, Y2.shape
```

```
(torch.Size([2, 55, 20, 20]), torch.Size([2, 33, 10, 10]))
```

```
def forward(x, block):
    block.initialize()
    return block(x)

Y1 = forward(np.zeros((2, 8, 20, 20)), cls_predictor(5, 10))
Y2 = forward(np.zeros((2, 16, 10, 10)), cls_predictor(3, 10))
Y1.shape, Y2.shape
```

```
[07:56:14] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
((2, 55, 20, 20), (2, 33, 10, 10))
```

```
batch_size = 32
train_iter, _ = d2l.load_data_bananas(batch_size)
```

```
batch_size = 32
train_iter, _ = d2l.load_data_bananas(batch_size)
```

見てのとおり、バッチサイズの次元を除けば、他の 3 つの次元はすべて異なるサイズを持っている。これら 2 つの予測出力をより効率的に計算するために連結するには、これらのテンソルをより一貫した形式に変換する。

チャネル次元には、同じ中心を持つアンカーボックスに対する予測が格納されていることに注意しよ。まずこの次元を最内側に移する。バッチサイズは異なるスケールでも同じなので、予測出力を（バッチサイズ，高さ × 幅 × チャネル数）という形状の 2 次元テンソルに変換できる。その後、異なるスケールのこのような出力を次元 1 に沿って連結できる。

```
def flatten_pred(pred):
    return torch.flatten(pred.permute(0, 2, 3, 1), start_dim=1)

def concat_preds(preds):
    return torch.cat([flatten_pred(p) for p in preds], dim=1)
```

```
def flatten_pred(pred):
    return npx.batch_flatten(pred.transpose(0, 2, 3, 1))

def concat_preds(preds):
    return np.concatenate([flatten_pred(p) for p in preds], axis=1)
```

このようにして、`Y1` と `Y2` はチャネル数、高さ、幅が異なっていても、同じミニバッチに対する 2 つの異なるスケールの予測出力を連結できる。

```
concat_preds([Y1, Y2]).shape
```

```
torch.Size([2, 25300])
```

ダウンサンプリングブロック

複数スケールの物体を検出するために、入力特徴マップの高さと幅を半分にする以下のダウンサンプリングブロック `down_sample_blk` を定義する。実際には、このブロックは Section 11.2.1 の VGG ブロックの設計を適用している。より具体的には、各ダウンサンプリングブロックはパディング 1 の 3×3 畳み込み層 2 つと、ストライド 2 の 2×2 最大プーリング層 1 つから構成される。ご存じのように、パディング 1 の 3×3 畳み込み層は特徴マップの形状を変えない。しかし、その後に続く 2×2 最大プーリングは入力特徴マップの高さと幅を半分にする。このダウンサンプリングブロックの入力特徴マップと出力特徴マップの両方について、$1 \times 2 + (3 - 1) + (3 - 1) = 6$ であるため、出力の各ユニットは入力上で 6×6 の受容野を持つ。したがって、ダウンサンプリングブロックは出力特徴マップの各ユニットの受容野を拡大する。

```
def down_sample_blk(in_channels, out_channels):
    blk = []
    for _ in range(2):
        blk.append(nn.Conv2d(in_channels, out_channels,
                             kernel_size=3, padding=1))
        blk.append(nn.BatchNorm2d(out_channels))
        blk.append(nn.ReLU())
        in_channels = out_channels
    blk.append(nn.MaxPool2d(2))
    return nn.Sequential(*blk)
```

```
def down_sample_blk(num_channels):
    blk = nn.Sequential()
    for _ in range(2):
        blk.add(nn.Conv2D(num_channels, kernel_size=3, padding=1),
                nn.BatchNorm(in_channels=num_channels),
                nn.Activation('relu'))
    blk.add(nn.MaxPool2D(2))
    return blk
```

以下の例では、構成したダウンサンプリングブロックは入力チャネル数を変え、入力特徴マップの高さと幅を半分にする。

```
forward(torch.zeros((2, 3, 20, 20)), down_sample_blk(3, 10)).shape
```

```
torch.Size([2, 10, 10, 10])
```

```
forward(np.zeros((2, 3, 20, 20)), down_sample_blk(10)).shape
```

```
(2, 10, 10, 10)
```

ベースネットワークブロック

ベースネットワークブロックは入力画像から特徴を抽出するために使われる。簡単のために、3 つのダウンサンプリングブロックからなる小さなベースネットワークを構成し、各ブロックでチャネル数を 2 倍にする。256×256 の入力画像に対して、このベースネットワークブロックは 32×32 の特徴マップを出力する（$256/2^3 = 32$）。

```
def base_net():
    blk = []
    num_filters = [3, 16, 32, 64]
    for i in range(len(num_filters) - 1):
        blk.append(down_sample_blk(num_filters[i], num_filters[i+1]))
    return nn.Sequential(*blk)

forward(torch.zeros((2, 3, 256, 256)), base_net()).shape
```

```
torch.Size([2, 64, 32, 32])
```

```
def base_net():
    blk = nn.Sequential()
    for num_filters in [16, 32, 64]:
        blk.add(down_sample_blk(num_filters))
    return blk

forward(np.zeros((2, 3, 256, 256)), base_net()).shape
```

```
(2, 64, 32, 32)
```

完全なモデル

完全なシングルショット・マルチボックス検出モデルは 5 つのブロックから構成される。各ブロックで生成された特徴マップは、(i) アンカーボックスの生成と、(ii) それらのアンカーボックスのクラスとオフセットの予測の両方に使われる。この 5 つのブロックのうち、最初のものがベースネットワークブロック、2 番目から 4 番目がダウンサンプリングブロック、最後のブロックはグローバル最大プーリングを使って高さと幅の両方を 1 に縮小する。技術的には、2 番目から 5 番目のブロックはすべて Fig. 17.7.1 におけるマルチスケール特徴マップブロックである。

```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 1:
        blk = down_sample_blk(64, 128)
    elif i == 4:
        blk = nn.AdaptiveMaxPool2d((1,1))
    else:
        blk = down_sample_blk(128, 128)
    return blk
```

```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 4:
        blk = nn.GlobalMaxPool2D()
    else:
        blk = down_sample_blk(128)
    return blk
```

ここで、各ブロックの順伝播を定義する。画像分類タスクとは異なり、ここでの出力には (i) CNN 特徴マップ Y、(ii) 現在のスケールで Y を用いて生成されたアンカーボックス、(iii) これらのアンカーボックスに対して Y に基づいて予測されたクラスとオフセットが含まれる。

```
def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = d2l.multibox_prior(Y, sizes=size, ratios=ratio)
    cls_preds = cls_predictor(Y)
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)
```

Fig. 17.7.1 で見たように、上部に近いマルチスケール特徴マップブロックほどより大きな物体の検出に使われるため、より大きなアンカーボックスを生成する必要がある。上の順伝播では、各マルチスケール特徴マップブロックにおいて、呼び出される `multibox_prior` 関数（Section 17.4 で説明）の `size` 引数を通じて 2 つのスケール値のリストを渡している。以下では、0.2 と 1.05 の間の区間を 5 つの区間に等分し、5 つのブロックにおける小さいスケール値を 0.2, 0.37, 0.54, 0.71, 0.88 と決める。その後、それらの大きいスケール値は $\sqrt{0.2 \times 0.37} = 0.272$, $\sqrt{0.37 \times 0.54} = 0.447$ などで与えられる。

```
sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
         [0.88, 0.961]]
ratios = [[1, 2, 0.5]] * 5
num_anchors = len(sizes[0]) + len(ratios[0]) - 1
```

これで、以下のように完全なモデル `TinySSD` を定義できる。

```
class TinySSD(nn.Module):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        idx_to_in_channels = [64, 128, 128, 128, 128]
        for i in range(5):
            # Equivalent to the assignment statement `self.blk_i = get_blk(i)`
            setattr(self, f'blk_{i}', get_blk(i))
            setattr(self, f'cls_{i}', cls_predictor(idx_to_in_channels[i],
                                                    num_anchors, num_classes))
            setattr(self, f'bbox_{i}', bbox_predictor(idx_to_in_channels[i],
                                                      num_anchors))

    def forward(self, X):
        anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
        for i in range(5):
            # Here `getattr(self, 'blk_%d' % i)` accesses `self.blk_i`
            X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
                X, getattr(self, f'blk_{i}'), sizes[i], ratios[i],
                getattr(self, f'cls_{i}'), getattr(self, f'bbox_{i}'))
        anchors = torch.cat(anchors, dim=1)
        cls_preds = concat_preds(cls_preds)
        cls_preds = cls_preds.reshape(
            cls_preds.shape[0], -1, self.num_classes + 1)
        bbox_preds = concat_preds(bbox_preds)
        return anchors, cls_preds, bbox_preds
```

```
class TinySSD(nn.Block):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        for i in range(5):
            # Equivalent to the assignment statement `self.blk_i = get_blk(i)`
            setattr(self, f'blk_{i}', get_blk(i))
            setattr(self, f'cls_{i}', cls_predictor(num_anchors, num_classes))
            setattr(self, f'bbox_{i}', bbox_predictor(num_anchors))

    def forward(self, X):
        anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
        for i in range(5):
            # Here `getattr(self, 'blk_%d' % i)` accesses `self.blk_i`
            X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
                X, getattr(self, f'blk_{i}'), sizes[i], ratios[i],
                getattr(self, f'cls_{i}'), getattr(self, f'bbox_{i}'))
        anchors = np.concatenate(anchors, axis=1)
        cls_preds = concat_preds(cls_preds)
        cls_preds = cls_preds.reshape(
            cls_preds.shape[0], -1, self.num_classes + 1)
        bbox_preds = concat_preds(bbox_preds)
        return anchors, cls_preds, bbox_preds
```

ここで、256×256 画像のミニバッチ `X` に対して順伝播を行うためにモデルインスタンスを作成する。

この節の前半で示したように、最初のブロックは 32×32 の特徴マップを出力する。2 番目から 4 番目のダウンサンプリングブロックは高さと幅を半分にし、5 番目のブロックはグローバルプーリングを使うことを思い出しよ。特徴マップの空間次元の各ユニットに対して 4 個のアンカーボックスが生成されるので、5 つのスケール全体では各画像に対して合計 $(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$ 個のアンカーボックスが生成される。

```
net = TinySSD(num_classes=1)
X = torch.zeros((32, 3, 256, 256))
anchors, cls_preds, bbox_preds = net(X)

print('output anchors:', anchors.shape)
print('output class preds:', cls_preds.shape)
print('output bbox preds:', bbox_preds.shape)
```

```
output anchors: torch.Size([1, 5444, 4])
output class preds: torch.Size([32, 5444, 2])
output bbox preds: torch.Size([32, 21776])
```

```
net = TinySSD(num_classes=1)
net.initialize()
X = np.zeros((32, 3, 256, 256))
anchors, cls_preds, bbox_preds = net(X)
```

(continues on next page)

(continued from previous page)

```
print('output anchors:', anchors.shape)
print('output class preds:', cls_preds.shape)
print('output bbox preds:', bbox_preds.shape)
```

```
output anchors: (1, 5444, 4)
output class preds: (32, 5444, 2)
output bbox preds: (32, 21776)
```

17.7.2 学習

ここでは、物体検出のためにシングルショット・マルチボックス検出モデルをどのように学習するかを説明する。

データセットの読み込みとモデルの初期化

まず、sec_object-detection-dataset で説明したバナナ検出データセットを読み込みます。

```
batch_size = 32
train_iter, _ = d2l.load_data_bananas(batch_size)
```

```
read 1000 training examples
read 100 validation examples
```

バナナ検出データセットには 1 つのクラスしかない。モデルを定義した後は、そのパラメータを初期化し、最適化アルゴリズムを定義する必要がある。

```
device, net = d2l.try_gpu(), TinySSD(num_classes=1)
trainer = torch.optim.SGD(net.parameters(), lr=0.2, weight_decay=5e-4)
```

```
device, net = d2l.try_gpu(), TinySSD(num_classes=1)
net.initialize(init=init.Xavier(), ctx=device)
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'learning_rate': 0.2, 'wd': 5e-4})
```

```
[07:56:23] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

損失関数と評価関数の定義

物体検出には 2 種類の損失がある。1 つ目の損失はアンカーボックスのクラスに関するものである。その計算には、画像分類で使った交差エントロピー損失関数をそのまま再利用できる。2 つ目の損失は正例（背景でない）アンカーボ

ックスのオフセットに関するものである。これは回帰問題である。この回帰問題では、しかし、ここでは Section 6.1.3 で説明した二乗損失は使わない。代わりに、ℓ_1 ノルム損失、すなわち予測値と正解値の差の絶対値を使う。マスク変数 bbox_masks は、損失計算において負例アンカーボックスと不正な（パディングされた）アンカーボックスを除外する。最後に、アンカーボックスのクラス損失とアンカーボックスのオフセット損失を足し合わせて、モデルの損失関数を得る。

```
cls_loss = nn.CrossEntropyLoss(reduction='none')
bbox_loss = nn.L1Loss(reduction='none')

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    batch_size, num_classes = cls_preds.shape[0], cls_preds.shape[2]
    cls = cls_loss(cls_preds.reshape(-1, num_classes),
                   cls_labels.reshape(-1)).reshape(batch_size, -1).mean(dim=1)
    bbox = bbox_loss(bbox_preds * bbox_masks,
                     bbox_labels * bbox_masks).mean(dim=1)
    return cls + bbox
```

```
cls_loss = gluon.loss.SoftmaxCrossEntropyLoss()
bbox_loss = gluon.loss.L1Loss()

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    cls = cls_loss(cls_preds, cls_labels)
    bbox = bbox_loss(bbox_preds * bbox_masks, bbox_labels * bbox_masks)
    return cls + bbox
```

分類結果の評価には精度を使える。オフセットに対して ℓ_1 ノルム損失を使っているため、予測されたバウンディングボックスの評価には 平均絶対誤差を使う。これらの予測結果は、生成されたアンカーボックスとそれらに対する予測オフセットから得られる。

```
def cls_eval(cls_preds, cls_labels):
    # Because the class prediction results are on the final dimension,
    # `argmax` needs to specify this dimension
    return float((cls_preds.argmax(dim=-1).type(
        cls_labels.dtype) == cls_labels).sum())

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((torch.abs((bbox_labels - bbox_preds) * bbox_masks)).sum())
```

```
def cls_eval(cls_preds, cls_labels):
    # Because the class prediction results are on the final dimension,
    # `argmax` needs to specify this dimension
    return float((cls_preds.argmax(axis=-1).astype(
        cls_labels.dtype) == cls_labels).sum())

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((np.abs((bbox_labels - bbox_preds) * bbox_masks)).sum())
```

モデルの学習

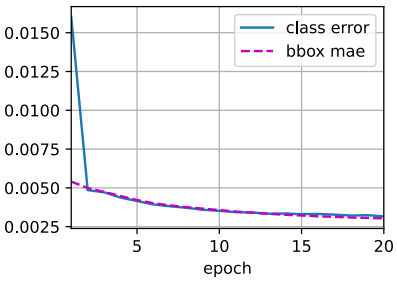
モデルを学習するときには、順伝播でマルチスケールのアンカーボックス (anchors) を生成し、それらのクラス (cls_preds) とオフセット (bbox_preds) を予測する必要がある。次に、ラベル情報 Y に基づいて、このように生成されたアンカーボックスのクラス (cls_labels) とオフセット (bbox_labels) にラベル付けする。最後に、クラスとオフセットの予測値とラベル値を使って損失関数を計算する。簡潔な実装のため、ここではテストデータセットの評価は省略する。

```
num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                        legend=['class error', 'bbox mae'])
net = net.to(device)
for epoch in range(num_epochs):
    # Sum of training accuracy, no. of examples in sum of training accuracy,
    # Sum of absolute error, no. of examples in sum of absolute error
    metric = d2l.Accumulator(4)
    net.train()
    for features, target in train_iter:
        timer.start()
        trainer.zero_grad()
        X, Y = features.to(device), target.to(device)
        # Generate multiscale anchor boxes and predict their classes and
        # offsets
        anchors, cls_preds, bbox_preds = net(X)
        # Label the classes and offsets of these anchor boxes
        bbox_labels, bbox_masks, cls_labels = d2l.multibox_target(anchors, Y)
        # Calculate the loss function using the predicted and labeled values
        # of the classes and offsets
        l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                      bbox_masks)
        l.mean().backward()
        trainer.step()
        metric.add(cls_eval(cls_preds, cls_labels), cls_labels.numel(),
                   bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                   bbox_labels.numel())
    cls_err, bbox_mae = 1 - metric[0] / metric[1], metric[2] / metric[3]
    animator.add(epoch + 1, (cls_err, bbox_mae))
print(f'class err {cls_err:.2e}, bbox mae {bbox_mae:.2e}')
print(f'{len(train_iter.dataset) / timer.stop():.1f} examples/sec on '
      f'{str(device)}')
```

```
class err 3.17e-03, bbox mae 3.03e-03
4427.3 examples/sec on cuda:0
```

```
num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                        legend=['class error', 'bbox mae'])
for epoch in range(num_epochs):
    # Sum of training accuracy, no. of examples in sum of training accuracy,
    # Sum of absolute error, no. of examples in sum of absolute error
```

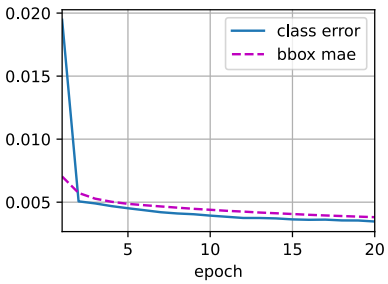
(continues on next page)



(continued from previous page)

```
    metric = d2l.Accumulator(4)
    for features, target in train_iter:
        timer.start()
        X = features.as_in_ctx(device)
        Y = target.as_in_ctx(device)
        with autograd.record():
            # Generate multiscale anchor boxes and predict their classes and
            # offsets
            anchors, cls_preds, bbox_preds = net(X)
            # Label the classes and offsets of these anchor boxes
            bbox_labels, bbox_masks, cls_labels = d2l.multibox_target(anchors,
                                                                      Y)
            # Calculate the loss function using the predicted and labeled
            # values of the classes and offsets
            l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                          bbox_masks)
        l.backward()
        trainer.step(batch_size)
        metric.add(cls_eval(cls_preds, cls_labels), cls_labels.size,
                   bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                   bbox_labels.size)
    cls_err, bbox_mae = 1 - metric[0] / metric[1], metric[2] / metric[3]
    animator.add(epoch + 1, (cls_err, bbox_mae))
print(f'class err {cls_err:.2e}, bbox mae {bbox_mae:.2e}')
print(f'{len(train_iter._dataset) / timer.stop():.1f} examples/sec on '
      f'{str(device)}')
```

```
class err 3.47e-03, bbox mae 3.81e-03
2046.8 examples/sec on gpu(0)
```



17.7.3 予測

予測時の目標は、画像上の関心対象のすべての物体を検出することである。以下では、テスト画像を読み込み、リサイズして、畳み込み層に必要な 4 次元テンソルに変換する。

```
X = torchvision.io.read_image('../img/banana.jpg').unsqueeze(0).float()
img = X.squeeze(0).permute(1, 2, 0).long()
```

```
img = image.imread('../img/banana.jpg')
feature = image.imresize(img, 256, 256).astype('float32')
X = np.expand_dims(feature.transpose(2, 0, 1), axis=0)
```

以下の multibox_detection 関数を使うと、予測されたバウンディングボックスはアンカーボックスとそれらの予測オフセットから得られる。その後、非最大抑制を使って類似した予測バウンディングボックスを除去する。

```
def predict(X):
    net.eval()
    anchors, cls_preds, bbox_preds = net(X.to(device))
    cls_probs = F.softmax(cls_preds, dim=2).permute(0, 2, 1)
    output = d2l.multibox_detection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
    return output[0, idx]

output = predict(X)
```

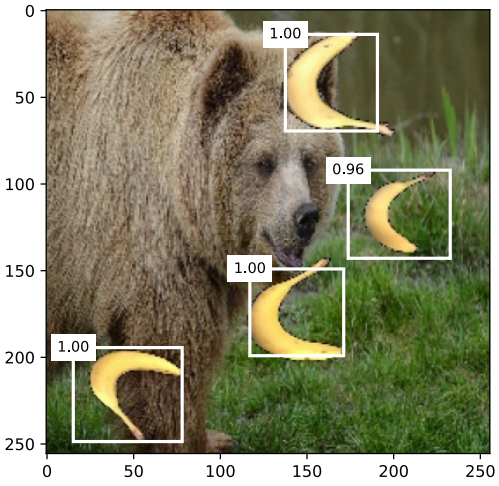
```
def predict(X):
    anchors, cls_preds, bbox_preds = net(X.as_in_ctx(device))
    cls_probs = npx.softmax(cls_preds).transpose(0, 2, 1)
    output = d2l.multibox_detection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
    return output[0, idx]

output = predict(X)
```

最後に、信頼度が 0.9 以上のすべての予測バウンディングボックスを表示する。

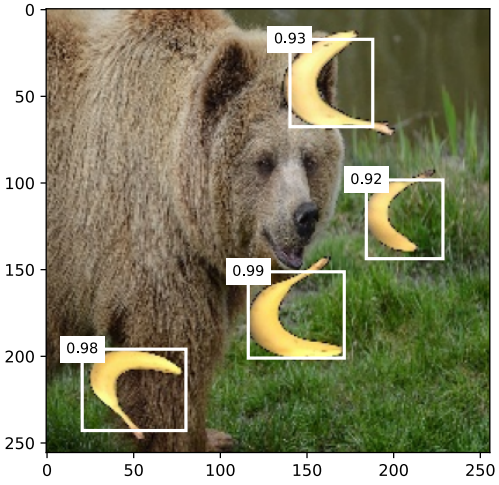
```
def display(img, output, threshold):
    d2l.set_figsize((5, 5))
    fig = d2l.plt.imshow(img)
    for row in output:
        score = float(row[1])
        if score < threshold:
            continue
        h, w = img.shape[:2]
        bbox = [row[2:6] * torch.tensor((w, h, w, h), device=row.device)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output.cpu(), threshold=0.9)
```




```
def display(img, output, threshold):
    d2l.set_figsize((5, 5))
    fig = d2l.plt.imshow(img.asnumpy())
    for row in output:
        score = float(row[1])
        if score < threshold:
            continue
        h, w = img.shape[:2]
        bbox = [row[2:6] * np.array((w, h, w, h), ctx=row.ctx)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output, threshold=0.9)
```



17.7.4 まとめ

- シングルショット・マルチボックス検出はマルチスケール物体検出モデルである。ベースネットワークといくつかのマルチスケール特徴マップブロックを通じて、シングルショット・マルチボックス検出は異なるサイズを持つさまざまな数のアンカーボックスを生成し、これらのアンカーボックス（したがってバウンディングボックス）のクラスとオフセットを予測することでさまざまなサイズの物体を検出する。
- シングルショット・マルチボックス検出モデルを学習するとき、損失関数はアンカーボックスのクラスとオフセットの予測値とラベル値に基づいて計算される。

17.7.5 演習

1. 損失関数を改善することで、シングルショット・マルチボックス検出を改良できるか？たとえば、予測オフセットに対する ℓ_1 ノルム損失を smooth ℓ_1 ノルム損失に置き換えてみよ。この損失関数は、滑らかさのためにゼロ付近で二乗関数を使い、その形はハイパーパラメータ σ によって制御される。

$$f(x) = \begin{cases} (\sigma x)^2/2, & \text{if } |x| < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{otherwise} \end{cases} \tag{17.7.1}$$

σ が非常に大きいとき、この損失は ℓ_1 ノルム損失に似ている。値が小さいほど、損失関数はより滑らかになる。

```
def smooth_l1(data, scalar):
    out = []
    for i in data:
        if abs(i) < 1 / (scalar ** 2):
            out.append(((scalar * i) ** 2) / 2)
        else:
            out.append(abs(i) - 0.5 / (scalar ** 2))
    return torch.tensor(out)

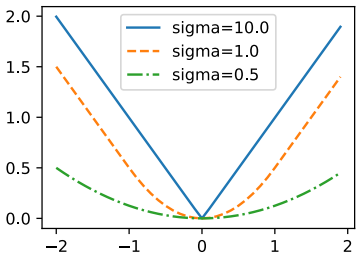
sigmas = [10, 1, 0.5]
lines = ['-', '--', '-.']
x = torch.arange(-2, 2, 0.1)
d2l.set_figsize()

for l, s in zip(lines, sigmas):
    y = smooth_l1(x, scalar=s)
    d2l.plt.plot(x, y, l, label='sigma=%.1f' % s)
d2l.plt.legend();
```

```
sigmas = [10, 1, 0.5]
lines = ['-', '--', '-.']
x = np.arange(-2, 2, 0.1)
d2l.set_figsize()

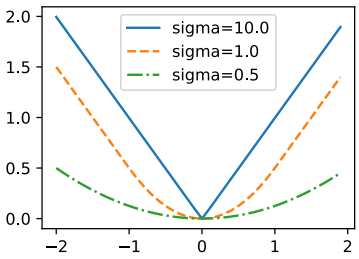
for l, s in zip(lines, sigmas):
    y = npx.smooth_l1(x, scalar=s)
```

(continues on next page)



(continued from previous page)

```
    d2l.plt.plot(x.asnumpy(), y.asnumpy(), l, label='sigma=%.1f' % s)
d2l.plt.legend();
```



さらに、この実験ではクラス予測に交差エントロピー損失を使った。正解クラス j に対する予測確率を p_j とすると、交差エントロピー損失は $-\log p_j$ である。focal loss (Lin *et al.*, 2017) も使える。ハイパーパラメータ $\gamma > 0$ と $\alpha > 0$ を用いると、この損失は次のように定義される。

$$-\alpha(1-p_j)^\gamma \log p_j. \tag{17.7.2}$$

見てのとおり、γ を増やすと、うまく分類された例（たとえば $p_j > 0.5$）に対する相対的な損失を効果的に減らせるので、学習は誤分類された難しい例により重点を置けるようになる。

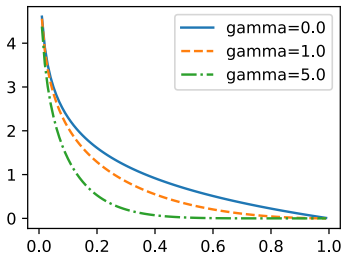
```
def focal_loss(gamma, x):
    return -(1 - x) ** gamma * torch.log(x)

x = torch.arange(0.01, 1, 0.01)
for l, gamma in zip(lines, [0, 1, 5]):
    y = d2l.plt.plot(x, focal_loss(gamma, x), l, label='gamma=%.1f' % gamma)
d2l.plt.legend();
```

```
def focal_loss(gamma, x):
    return -(1 - x) ** gamma * np.log(x)

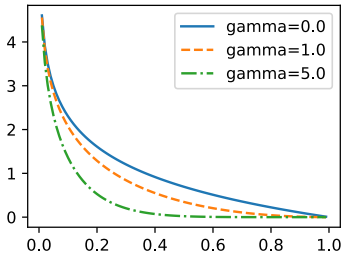
x = np.arange(0.01, 1, 0.01)
for l, gamma in zip(lines, [0, 1, 5]):
```

(continues on next page)



(continued from previous page)

```
    y = d2l.plt.plot(x.asnumpy(), focal_loss(gamma, x).asnumpy(), l,
                     label='gamma=%.1f' % gamma)
d2l.plt.legend();
```



2. 紙幅の制約のため、この節ではシングルショット・マルチボックス検出モデルのいくつかの実装詳細を省略した。次の観点でさらにモデルを改善できるか。

    1. 物体が画像に比べて非常に小さい場合、モデルは入力画像をより大きくリサイズできる。
    2. 通常、負例アンカーボックスは非常に多数ある。クラス分布をより均衡にするために、負例アンカーボックスをダウンサンプリングできる。
    3. 損失関数では、クラス損失とオフセット損失に異なる重みのハイパーパラメータを割り当てる。
    4. シングルショット・マルチボックス検出の論文 (Liu *et al.*, 2016) にあるような、他の方法で物体検出モデルを評価する。

17.8 Region-based CNNs (R-CNNs)

単発マルチボックス検出については Section 17.7 で説明したが、領域ベースの CNN、すなわち CNN 特徴を用いた領域（R-CNNs）も、物体検出に深層学習

を適用する先駆的な手法の一つです (Girshick *et al.*, 2014)。この節では、R-CNN とその一連の改良版である fast R-CNN (Girshick, 2015)、faster R-CNN (Ren *et al.*, 2015)、および mask R-CNN (He *et al.*, 2017) を紹介する。紙幅の都合上、これらのモデルの設計にのみ焦点を当てる。

17.8.1 R-CNNs

R-CNN はまず、入力画像から多数（たとえば 2000 個）の *region proposals* を抽出し（たとえば、アンカーボックスも region proposals と見なせます）、それらのクラスとバウンディングボックス（たとえばオフセット）をラベル付けする。

(Girshick *et al.*, 2014)

次に、各 region proposal に対して CNN を用いて順伝播を行い、その特徴を抽出す。その後、各 region proposal の特徴を用いて、この region proposal のクラスとバウンディングボックスを予測する。

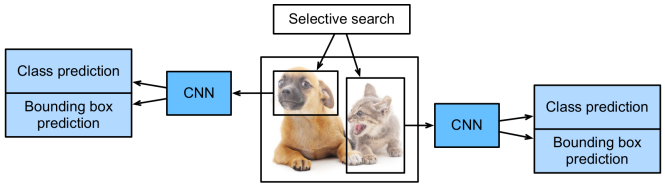


Fig. 17.8.1 The R-CNN model.

Fig. 17.8.1 は R-CNN モデルを示している。より具体的には、R-CNN は次の 4 つのステップから構成される。

1. 入力画像上で複数の高品質な region proposals を抽出するために、*selective search* を実行する (Uijlings *et al.*, 2013)。これらの提案領域は通常、異なる形状やサイズを持つ複数のスケールで選択される。各 region proposal には、クラスと正解バウンディングボックスがラベル付けされる。
2. 学習済み CNN を選び、出力層の手前で切り取りる。各 region proposal をネットワークが要求する入力サイズにリサイズし、順伝播によってその region proposal の抽出特徴を出力する。
3. 抽出した特徴と各 region proposal のラベル付きクラスを例として用いる。複数のサポートベクターマシンを学習して物体を分類し、各サポートベクターマシンはその例に特定のクラスが含まれるかどうかを個別に判定する。
4. 抽出した特徴と各 region proposal のラベル付きバウンディングボックスを例として用いる。線形回帰モデルを学習して、正解バウンディングボックスを予測する。

R-CNN モデルは学習済み CNN を用いて画像特徴を効果的に抽出すが、処理速度は遅いである。1 枚の入力画像から数千個の region proposals を選ぶことを考えてみよ。これには、物体検出を行うために数千回の CNN 順伝播が必要になる。こ

の膨大な計算負荷のため、R-CNN を実世界のアプリケーションで広く使うことは現実的ではない。

17.8.2 Fast R-CNN

R-CNN における主な性能上のボトルネックは、計算を共有せずに各 region proposal ごとに独立して CNN 順伝播を行う点にある。これらの領域は通常重なりを持つため、独立した特徴抽出では同じ計算が何度も繰り返される。*fast R-CNN* における R-CNN からの主要な改良点の一つは、CNN の順伝播を画像全体に対してのみ行うことです (Girshick, 2015)。

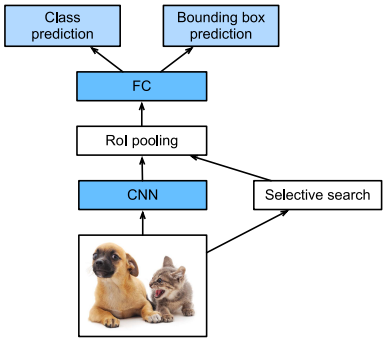


Fig. 17.8.2 The fast R-CNN model.

Fig. 17.8.2 は fast R-CNN モデルを示している。主な計算は次のとおりです。

1. R-CNN と比べると、fast R-CNN では特徴抽出のための CNN の入力は個々の region proposal ではなく、画像全体である。さらに、この CNN は学習可能である。入力画像が与えられたとき、CNN 出力の形状を $1 \times c \times h_1 \times w_1$ とする。

2. selective search が n 個の region proposals を生成すると仮定する。これらの region proposals（形状はさまざま）は、CNN 出力上の region of interest（形状はさまざま）を示す。次に、これらの region of interest から、同じ形状（たとえば高さ h_2 と幅 w_2 が指定される）の特徴をさらに抽出し、連結しやすくする。これを実現するために、fast R-CNN は *region of interest (RoI) pooling* 層を導入する。CNN 出力と region proposals をこの層に入力し、形状 $n \times c \times h_2 \times w_2$ の連結特徴を出力して、すべての region proposals に対してさらに特徴抽出を行う。

3. 全結合層を用いて、連結特徴を形状 $n \times d$ の出力に変換する。ここで d はモデル設計に依存する。

4. n 個の region proposals それぞれについて、クラスとバウンディングボックスを予測する。より具体的には、クラス予測とバウンディングボックス予測において、全結合層の出力をそれぞれ形状 $n \times q$（q はクラス数）と形状 $n \times 4$ の出力に変換する。クラス予測には softmax 回帰を用いる。

fast R-CNN で提案された region of interest pooling 層は、Section 10.5 で導入した pooling 層とは異なる。pooling 層では、pooling ウィンドウ、パディング、スト

ライドのサイズを指定することで、出力形状を間接的に制御する。これに対して、region of interest pooling 層では出力形状を直接指定できる。

たとえば、各領域の出力高さと幅をそれぞれ h_2 と w_2 に指定するとする。形状 $h \times w$ の任意の region of interest ウィンドウに対して、このウィンドウは $h_2 \times w_2$ のグリッドのサブウィンドウに分割され、各サブウィンドウの形状はおおよそ $(h/h_2) \times (w/w_2)$ になる。実際には、任意のサブウィンドウの高さと幅は切り上げられ、最大要素がサブウィンドウの出力として用いられる。したがって、region of interest pooling 層は、region of interest の形状が異なっていても同じ形状の特徴を抽出できる。

説明のための例として、Fig. 17.8.3 では、4×4 の入力上で左上の 3×3 の region of interest が選択されている。この region of interest に対して、2×2 の region of interest pooling 層を用いて 2×2 の出力を得る。4 つに分割された各サブウィンドウには要素 0, 1, 4, 5（最大値は 5）、2, 6（最大値は 6）、8, 9（最大値は 9）、および 10 が含まれることに注意しよ。

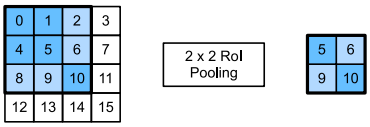


Fig. 17.8.3 A 2×2 region of interest pooling layer.

以下では、region of interest pooling 層の計算を示す。CNN で抽出された特徴 X の高さと幅がともに 4 で、チャネルが 1 つだけであると仮定する。

```
import torch
import torchvision

X = torch.arange(16.).reshape(1, 1, 4, 4)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

```
from mxnet import np, npx

npx.set_np()

X = np.arange(16).reshape(1, 1, 4, 4)
X
```

```
[07:02:53] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
array([[[[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.]]])
```

さらに、入力画像の高さと幅がともに 40 ピクセルであり、selective search がこの画像上で 2 つの region proposals を生成すると仮定しよう。各 region proposal は 5 つの要素で表される。すなわち、物体クラスに続いて、その左上隅と右下隅の (x, y) 座標である。

```
rois = torch.Tensor([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]])
```

```
rois = np.array([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]])
```

`X` の高さと幅は入力画像の高さと幅の 1/10 なので、2 つの region proposals の座標は、指定された `spatial_scale` 引数に従って 0.1 倍される。すると、2 つの region of interest はそれぞれ `X[:, :, 0:3, 0:3]` と `X[:, :, 1:4, 0:4]` として `X` 上に示される。最後に、2×2 の region of interest pooling では、各 region of interest はサブウィンドウのグリッドに分割され、同じ形状 2×2 の特徴をさらに抽出す。

```
torchvision.ops.roi_pool(X, rois, output_size=(2, 2), spatial_scale=0.1)
```

```
tensor([[[[ 5.,  6.],
          [ 9., 10.]]],


        [[[ 9., 11.],
          [13., 15.]]]])
```

```
npx.roi_pooling(X, rois, pooled_size=(2, 2), spatial_scale=0.1)
```

```
array([[[[ 5.,  6.],
         [ 9., 10.]]],


       [[[ 9., 11.],
         [13., 15.]]]])
```

17.8.3 Faster R-CNN

物体検出をより高精度にするために、fast R-CNN モデルでは通常、selective search によって多数の region proposals を生成する必要がある。精度を落とさずに region proposals を減らすために、*faster R-CNN* は selective search を *region proposal network* に置き換えることを提案した (Ren *et al.*, 2015)。

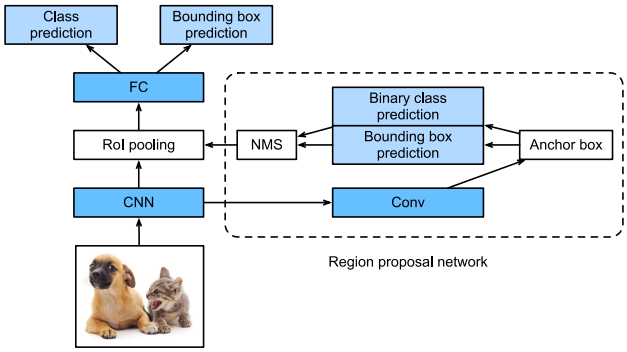


Fig. 17.8.4 The faster R-CNN model.

Fig. 17.8.4 は faster R-CNN モデルを示している。fast R-CNN と比べると、faster R-CNN が変更するのは region proposal の方法を selective search から region proposal network に変える点だけである。モデルの残りの部分は変更されない。region proposal network は次の手順で動作する。

1. パディング 1 の 3×3 畳み込み層を用いて、CNN 出力を c チャネルの新しい出力に変換する。これにより、CNN で抽出された特徴マップの空間次元上の各ユニットは、長さ c の新しい特徴ベクトルを持つようになる。
2. 特徴マップの各ピクセルを中心として、異なるスケールとアスペクト比を持つ複数のアンカーボックスを生成し、それらにラベルを付ける。
3. 各アンカーボックスの中心にある長さ c の特徴ベクトルを用いて、二値クラス（背景または物体）と、このアンカーボックスのバウンディングボックスを予測する。
4. 予測されたクラスが物体であるバウンディングボックスを考える。non-maximum suppression を用いて重なりのある結果を除去する。残った物体の予測バウンディングボックスが、region of interest pooling 層に必要な region proposals である。

注目すべき点として、faster R-CNN モデルの一部として、region proposal network はモデルの残りの部分と同時に学習される。言い換えると、faster R-CNN の目的関数には、物体検出におけるクラスとバウンディングボックスの予測だけでなく、region proposal network におけるアンカーボックスの二値クラスとバウンディングボックスの予測も含まれる。エンドツーエンド学習の結果として、region proposal network は高品質な region proposals を生成する方法を学習し、データから学習された少数の region proposals で物体検出の精度を保てるようになる。

17.8.4 Mask R-CNN

学習データセットにおいて、画像上の物体のピクセルレベルの位置もラベル付けされているなら、*mask R-CNN* はそのような詳細なラベルを効果的に活用して、物体検出の精度をさらに向上できる (He *et al.*, 2017)。

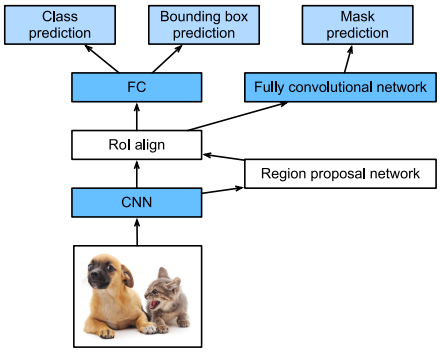


Fig. 17.8.5 The mask R-CNN model.

Fig. 17.8.5 に示すように、mask R-CNN は faster R-CNN を基に改良されている。具体的には、mask R-CNN は region of interest pooling 層を *region of interest (RoI) alignment* 層に置き換える。この region of interest alignment 層は双線形補間を用いて特徴マップ上の空間情報を保持し、ピクセルレベルの予測により適している。この層の出力には、すべての region of interest に対して同じ形状の特徴マップが含まれる。それらは、各 region of interest のクラスとバウンディングボックスを予測するだけでなく、追加の全畳み込みネットワークを通じて物体のピクセルレベルの位置も予測するために用いられる。画像のピクセルレベルの意味情報を予測するために全畳み込みネットワークを用いる詳細については、この章の後続の節で説明する。

17.8.5 要約

- R-CNN は入力画像から多数の region proposals を抽出し、CNN を用いて各 region proposal に順伝播を行って特徴を抽出し、その特徴を用いてこの region proposal のクラスとバウンディングボックスを予測する。
- fast R-CNN における R-CNN からの主要な改良点の一つは、CNN の順伝播を画像全体に対してのみ行うことである。また、region of interest pooling 層を導入し、形状の異なる region of interest に対して同じ形状の特徴をさらに抽出できるようにしている。
- faster R-CNN は、fast R-CNN で用いられる selective search を同時学習される region proposal network に置き換えることで、少数の region proposals でも物体検出の精度を保てるようにしている。
- faster R-CNN を基に、mask R-CNN はさらに全畳み込みネットワークを導入し、ピクセルレベルのラベルを活用して物体検出の精度をさらに向上させる。

17.8.6 演習

1. 物体検出を、バウンディングボックスとクラス確率の予測のような単一の回帰問題として定式化できるだろうか。YOLO モデルの設計を参照してもよいです (Redmon *et al.*, 2016)。
2. 単発マルチボックス検出と、この節で紹介した手法を比較しよ。主な違いは何だろうか。Zhao *et al.* (2019) の図 2 を参照してもよいである。

17.9 セマンティックセグメンテーションとデータセット

Section 17.3–Section 17.8 で物体検出タスクについて述べたときには、画像中の物体をラベル付けして予測するために矩形のバウンディングボックスを用いた。この節では、画像を異なる意味クラスに属する領域へどのように分割するかに焦点を当てる セマンティックセグメンテーションの問題を扱いる。物体検出とは異なり、セマンティックセグメンテーションは画像中に何があるかをピクセルレベルで認識し理解する。つまり、そのラベル付けと意味領域の予測はピクセルレベルで行われる。Fig. 17.9.1 は、セマンティックセグメンテーションにおける画像の犬、猫、背景のラベルを示している。物体検出と比べると、セマンティックセグメンテーションでラベル付けされるピクセルレベルの境界は、明らかにより細かい粒度である。

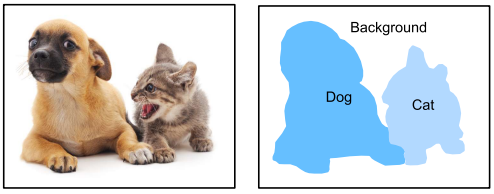


Fig. 17.9.1 セマンティックセグメンテーションにおける画像の犬、猫、背景のラベル。

17.9.1 画像セグメンテーションとインスタンスセグメンテーション

セマンティックセグメンテーションに似た、コンピュータビジョン分野の重要なタスクとして、画像セグメンテーションとインスタンスセグメンテーションもある。以下では、これらをセマンティックセグメンテーションと簡単に区別する。

- 画像セグメンテーションは、画像をいくつかの構成領域に分割する。この種の問題の手法は通常、画像内のピクセル間の相関を利用する。学習時に画像ピクセルに関するラベル情報を必要とせず、予測時に得たい意味を持つ領域が分割されることを保証できない。Fig. 17.9.1 の画像を入力とすると、画像セグメンテーションは犬を 2 つの領域に分けるかもしれない。1 つは主

に黒い口と目を含む領域で、もう 1 つは主に黄色い残りの体を含む領域である。

- インスタンスセグメンテーションは、同時検出とセグメンテーションとも呼ばれる。画像中の各物体インスタンスのピクセルレベルの領域をどのように認識するかを研究する。セマンティックセグメンテーションとは異なり、インスタンスセグメンテーションでは意味だけでなく、異なる物体インスタンスも区別する必要がある。たとえば、画像に 2 匹の犬がいる場合、インスタンスセグメンテーションでは、あるピクセルが 2 匹のうちどちらの犬に属するかを区別する必要がある。

17.9.2 Pascal VOC2012 セマンティックセグメンテーションデータセット

111 

最も重要なセマンティックセグメンテーションデータセットの 1 つは Pascal VOC2012[111] である。以下では、このデータセットを見ていきる。

```
%matplotlib inline
from d2l import torch as d2l
import torch
import torchvision
import os
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, np, npx
import os

npx.set_np()
```

```
#@save
d2l.DATA_HUB['voc2012'] = (d2l.DATA_URL + 'VOCtrainval_11-May-2012.tar',
                           '4e443f8a2eca6b1dac8a6c57641b67dd40621a49')

voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
```

```
#@save
d2l.DATA_HUB['voc2012'] = (d2l.DATA_URL + 'VOCtrainval_11-May-2012.tar',
                           '4e443f8a2eca6b1dac8a6c57641b67dd40621a49')

voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
```

このデータセットの tar ファイルは約 2 GB あるため、ダウンロードにはしばらく時間がかかるかもしれない。展開後のデータセットは `../data/VOCdevkit/VOC2012` にある。

```
#@save
d2l.DATA_HUB['voc2012'] = (d2l.DATA_URL + 'VOCtrainval_11-May-2012.tar',
                           '4e443f8a2eca6b1dac8a6c57641b67dd40621a49')

voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
```


```
#@save
d2l.DATA_HUB['voc2012'] = (d2l.DATA_URL + 'VOCtrainval_11-May-2012.tar',
                           '4e443f8a2eca6b1dac8a6c57641b67dd40621a49')

voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
```


```
Downloading ../data/VOCtrainval_11-May-2012.tar from http://d2l-data.s3-
↪accelerate.amazonaws.com/VOCtrainval_11-May-2012.tar...
```

```
#@save
VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
                [0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
                [64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
                [64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
                [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],
                [0, 64, 128]]

#@save
VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
               'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
               'diningtable', 'dog', 'horse', 'motorbike', 'person',
               'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```


```
#@save
VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
                [0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
                [64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
                [64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
                [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],
                [0, 64, 128]]

#@save
VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
               'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
               'diningtable', 'dog', 'horse', 'motorbike', 'person',
               'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```


../data/VOCdevkit/VOC2012 に入ると、データセットのさまざまな構成要素を確認できる。ImageSets/Segmentation パスには、学習サンプルとテストサンプルを指定するテキストファイルが含まれ、JPEGImages と SegmentationClass パスには、それぞれ各例の入力画像とラベルが保存されている。ここでのラベルも画像形式であり、ラベル付けされた入力画像と同じサイズである。さらに、任意のラベル画像において同じ色のピクセルは同じ意味クラスに属する。以下で

は、read_voc_images 関数を定義して、すべての入力画像とラベルをメモリに読み込みます。

```
#@save
def read_voc_images(voc_dir, is_train=True):
    """Read all VOC feature and label images."""
    txt_fname = os.path.join(voc_dir, 'ImageSets', 'Segmentation',
                             'train.txt' if is_train else 'val.txt')
    mode = torchvision.io.image.ImageReadMode.RGB
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    features, labels = [], []
    for i, fname in enumerate(images):
        features.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'JPEGImages', f'{fname}.jpg')))
        labels.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'SegmentationClass' ,f'{fname}.png'), mode))
    return features, labels

train_features, train_labels = read_voc_images(voc_dir, True)
```

```
#@save
def read_voc_images(voc_dir, is_train=True):
    """Read all VOC feature and label images."""
    txt_fname = os.path.join(voc_dir, 'ImageSets', 'Segmentation',
                             'train.txt' if is_train else 'val.txt')
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    features, labels = [], []
    for i, fname in enumerate(images):
        features.append(image.imread(os.path.join(
            voc_dir, 'JPEGImages', f'{fname}.jpg')))
        labels.append(image.imread(os.path.join(
            voc_dir, 'SegmentationClass', f'{fname}.png')))
    return features, labels

train_features, train_labels = read_voc_images(voc_dir, True)
```

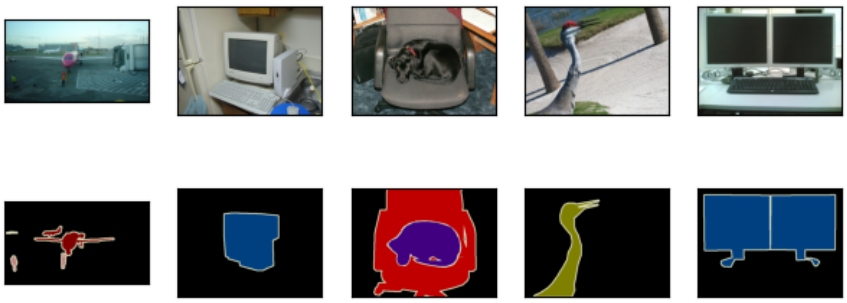
```
[07:02:12] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
y = voc_label_indices(train_labels[0], voc_colormap2label())
y[105:115, 130:140], VOC_CLASSES[1]
```

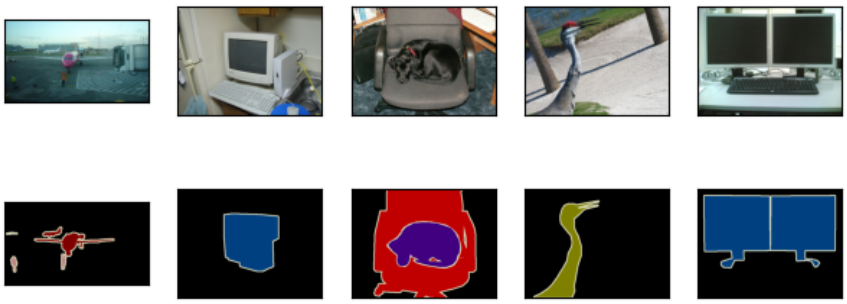
```
y = voc_label_indices(train_labels[0], voc_colormap2label())
y[105:115, 130:140], VOC_CLASSES[1]
```

最初の 5 つの入力画像とそのラベルを描画する。ラベル画像では、白と黒がそれぞれ境界と背景を表し、その他の色は異なるクラスに対応する。

```
n = 5
imgs = train_features[:n] + train_labels[:n]
imgs = [img.permute(1,2,0) for img in imgs]
d2l.show_images(imgs, 2, n);
```



```
n = 5
imgs = train_features[:n] + train_labels[:n]
d2l.show_images(imgs, 2, n);
```



```
crop_size = (320, 480)
voc_train = VOCSegDataset(True, crop_size, voc_dir)
voc_test = VOCSegDataset(False, crop_size, voc_dir)
```

```
crop_size = (320, 480)
voc_train = VOCSegDataset(True, crop_size, voc_dir)
voc_test = VOCSegDataset(False, crop_size, voc_dir)
```

次に、このデータセットのすべてのラベルについて、RGB の色値とクラス名を列挙する。

```
#@save
VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
                [0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
```

(continues on next page)

(continued from previous page)

```
                [64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
                [64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
                [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],
                [0, 64, 128]]

#@save
VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
               'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
               'diningtable', 'dog', 'horse', 'motorbike', 'person',
               'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```

上で定義した 2 つの定数を使うと、ラベル内の各ピクセルのクラスインデックスを簡単に求めることができる。voc_colormap2label 関数を定義して、上記の RGB 色値からクラスインデックスへの対応を構築し、voc_label_indices 関数を定義して、この Pascal VOC2012 データセットにおける任意の RGB 値をそのクラスインデックスに対応付ける。

```
#@save
def voc_colormap2label():
    """Build the mapping from RGB to class indices for VOC labels."""
    colormap2label = torch.zeros(256 ** 3, dtype=torch.long)
    for i, colormap in enumerate(VOC_COLORMAP):
        colormap2label[
            (colormap[0] * 256 + colormap[1]) * 256 + colormap[2]] = i
    return colormap2label

#@save
def voc_label_indices(colormap, colormap2label):
    """Map any RGB values in VOC labels to their class indices."""
    colormap = colormap.permute(1, 2, 0).numpy().astype('int32')
    idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256
           + colormap[:, :, 2])
    return colormap2label[idx]
```

```
#@save
def voc_colormap2label():
    """Build the mapping from RGB to class indices for VOC labels."""
    colormap2label = np.zeros(256 ** 3)
    for i, colormap in enumerate(VOC_COLORMAP):
        colormap2label[
            (colormap[0] * 256 + colormap[1]) * 256 + colormap[2]] = i
    return colormap2label

#@save
def voc_label_indices(colormap, colormap2label):
    """Map any RGB values in VOC labels to their class indices."""
    colormap = colormap.astype(np.int32)
    idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256
           + colormap[:, :, 2])
    return colormap2label[idx]
```

たとえば、最初の例画像では、飛行機の前方部分のクラスインデックスは 1 であり、背景のインデックスは 0 である。

```
y = voc_label_indices(train_labels[0], voc_colormap2label())
y[105:115, 130:140], VOC_CLASSES[1]
```

```
(tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 0, 0, 1, 1]]),
 'aeroplane')
```

データ前処理

Section 11.1–Section 11.4 などの以前の実験では、画像はモデルが要求する入力形状に合わせるためにリサイズされていた。しかし、セマンティックセグメンテーションでは、これを行うと予測されたピクセルクラスを元の入力画像の形状に戻す必要がある。このようなリサイズは、特に異なるクラスを含むセグメント領域では不正確になりえる。この問題を避けるために、リサイズの代わりに画像を 固定形状にクロップする。具体的には、画像拡張としてランダムクロップを用いて、入力画像とラベルの同じ領域を切り出す。

```
#@save
def voc_rand_crop(feature, label, height, width):
    """Randomly crop both feature and label images."""
    rect = torchvision.transforms.RandomCrop.get_params(
        feature, (height, width))
    feature = torchvision.transforms.functional.crop(feature, *rect)
    label = torchvision.transforms.functional.crop(label, *rect)
    return feature, label
```

```
#@save
def voc_rand_crop(feature, label, height, width):
    """Randomly crop both feature and label images."""
    feature, rect = image.random_crop(feature, (width, height))
    label = image.fixed_crop(label, *rect)
    return feature, label
```

```
imgs = []
for _ in range(n):
    imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 300)
```

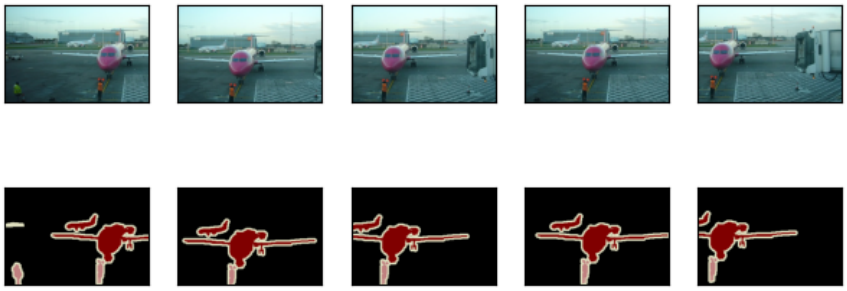
(continues on next page)

(continued from previous page)

```
imgs = [img.permute(1, 2, 0) for img in imgs]
d2l.show_images(imgs[::2] + imgs[1::2], 2, n);
```



```
imgs = []
for _ in range(n):
    imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 300)
d2l.show_images(imgs[::2] + imgs[1::2], 2, n);
```



カスタムセマンティックセグメンテーションデータセットクラス

高水準 API が提供する `Dataset` クラスを継承して、カスタムのセマンティックセグメンテーションデータセットクラス `VOCSegDataset` を定義する。`__getitem__` 関数を実装することで、データセット内で `idx` によってインデックス付けされた入力画像と、この画像の各ピクセルのクラスインデックスに任意にアクセスできる。データセット内の一部の画像はランダムクロップの出力サイズよりも小さいため、これらの例はカスタムの `filter` 関数によって除外される。さらに、入力画像の 3 つの RGB チャンネルの値を標準化するために `normalize_image` 関数も定義する。

```
#@save
class VOCSegDataset(torch.utils.data.Dataset):
    """A customized dataset to load the VOC dataset."""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, is_train, crop_size, voc_dir):
        self.transform = torchvision.transforms.Normalize(
            mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        self.crop_size = crop_size
        features, labels = read_voc_images(voc_dir, is_train=is_train)
        self.features = [self.normalize_image(feature)
                         for feature in self.filter(features)]
        self.labels = self.filter(labels)
        self.colormap2label = voc_colormap2label()
        print('read ' + str(len(self.features)) + ' examples')

    def normalize_image(self, img):
        return self.transform(img.float() / 255)

    def filter(self, imgs):
        return [img for img in imgs if (
            img.shape[1] >= self.crop_size[0] and
            img.shape[2] >= self.crop_size[1])]

    def __getitem__(self, idx):
        feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
                                       *self.crop_size)
        return (feature, voc_label_indices(label, self.colormap2label))

    def __len__(self):
        return len(self.features)
```

```
#@save
class VOCSegDataset(gluon.data.Dataset):
    """A customized dataset to load the VOC dataset."""
    def __init__(self, is_train, crop_size, voc_dir):
        self.rgb_mean = np.array([0.485, 0.456, 0.406])
        self.rgb_std = np.array([0.229, 0.224, 0.225])
        self.crop_size = crop_size
        features, labels = read_voc_images(voc_dir, is_train=is_train)
        self.features = [self.normalize_image(feature)
                         for feature in self.filter(features)]
        self.labels = self.filter(labels)
        self.colormap2label = voc_colormap2label()
        print('read ' + str(len(self.features)) + ' examples')

    def normalize_image(self, img):
        return (img.astype('float32') / 255 - self.rgb_mean) / self.rgb_std

    def filter(self, imgs):
        return [img for img in imgs if (
            img.shape[0] >= self.crop_size[0] and
            img.shape[1] >= self.crop_size[1])]

    def __getitem__(self, idx):
        feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
                                       *self.crop_size)
        return (feature.transpose(2, 0, 1),
                voc_label_indices(label, self.colormap2label))
```

(continues on next page)

(continued from previous page)

```
    def __len__(self):
        return len(self.features)
```

データセットの読み込み

カスタム VOCSegDataset クラスを用いて、学習セットとテストセットのインスタンスをそれぞれ作成する。ランダムクロップ後の出力形状を 320×480 と指定するとする。以下では、学習セットとテストセットに保持される例の数を確認できる。

```
crop_size = (320, 480)
voc_train = VOCSegDataset(True, crop_size, voc_dir)
voc_test = VOCSegDataset(False, crop_size, voc_dir)
```

```
read 1114 examples
read 1078 examples
```

バッチサイズを 64 に設定し、学習セットのデータイテレータを定義する。最初のミニバッチの形状を出力してみよう。画像分類や物体検出とは異なり、ここでのラベルは 3 次元テンソルである。

```
batch_size = 64
train_iter = torch.utils.data.DataLoader(voc_train, batch_size, shuffle=True,
                                         drop_last=True,
                                         num_workers=d2l.get_dataloader_workers())
for X, Y in train_iter:
    print(X.shape)
    print(Y.shape)
    break
```

```
torch.Size([64, 3, 320, 480])
torch.Size([64, 320, 480])
```

```
batch_size = 64
train_iter = gluon.data.DataLoader(voc_train, batch_size, shuffle=True,
                                   last_batch='discard',
                                   num_workers=d2l.get_dataloader_workers())
for X, Y in train_iter:
    print(X.shape)
    print(Y.shape)
    break
```

```
(64, 3, 320, 480)
(64, 320, 480)
```

全体をまとめる

最後に、Pascal VOC2012 セマンティックセグメンテーションデータセットをダウンロードして読み込むための load_data_voc 関数を定義する。これは学習データセットとテストデータセットの両方に対するデータイテレータを返す。

```
#@save
def load_data_voc(batch_size, crop_size):
    """Load the VOC semantic segmentation dataset."""
    voc_dir = d2l.download_extract('voc2012', os.path.join(
        'VOCdevkit', 'VOC2012'))
    num_workers = d2l.get_dataloader_workers()
    train_iter = torch.utils.data.DataLoader(
        VOCSegDataset(True, crop_size, voc_dir), batch_size,
        shuffle=True, drop_last=True, num_workers=num_workers)
    test_iter = torch.utils.data.DataLoader(
        VOCSegDataset(False, crop_size, voc_dir), batch_size,
        drop_last=True, num_workers=num_workers)
    return train_iter, test_iter
```

```
#@save
def load_data_voc(batch_size, crop_size):
    """Load the VOC semantic segmentation dataset."""
    voc_dir = d2l.download_extract('voc2012', os.path.join(
        'VOCdevkit', 'VOC2012'))
    num_workers = d2l.get_dataloader_workers()
    train_iter = gluon.data.DataLoader(
        VOCSegDataset(True, crop_size, voc_dir), batch_size,
        shuffle=True, last_batch='discard', num_workers=num_workers)
    test_iter = gluon.data.DataLoader(
        VOCSegDataset(False, crop_size, voc_dir), batch_size,
        last_batch='discard', num_workers=num_workers)
    return train_iter, test_iter
```

17.9.3 まとめ

- セマンティックセグメンテーションは、画像を異なる意味クラスに属する領域へ分割することで、画像中に何があるかをピクセルレベルで認識し理解する。
- 最も重要なセマンティックセグメンテーションデータセットの 1 つは Pascal VOC2012 である。
- セマンティックセグメンテーションでは、入力画像とラベルがピクセル単位で 1 対 1 に対応するため、入力画像はリサイズではなく、固定形状にランダムクロップされる。

17.9.4 演習

1. セマンティックセグメンテーションは、自動運転車や医用画像診断にどのように応用できるか？他の応用例も考えられますか？

2. Section 17.1 のデータ拡張の説明を思い出しよ。画像分類で用いられる画像拡張手法のうち、セマンティックセグメンテーションには適用できないものはどれだろうか。

17.10 転置畳み込み

これまで見てきた CNN の層、たとえば畳み込み層（Section 10.2）やプーリング層（Section 10.5）は、通常、入力の空間次元（高さと幅）を縮小（ダウンサンプリング）するか、あるいはそのまま維持する。画素レベルで分類を行うセマンティックセグメンテーションでは、入力と出力の空間次元が同じであると便利である。たとえば、ある出力画素のチャネル次元には、同じ空間位置にある入力画素の分類結果を保持できる。

これを実現するために、特に CNN 層によって空間次元が縮小された後では、中間特徴マップの空間次元を増やす（アップサンプリングする）別の種類の CNN 層を使うことができる。この節では、畳み込みによるダウンサンプリング操作を逆にするための、転置畳み込み（*fractionally-strided convolution* とも呼ばれる）(Dumoulin and Visin, 2016) を紹介する。

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
from mxnet import np, npx, init
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

```
def trans_conv(X, K):
    h, w = K.shape
    Y = d2l.zeros((X.shape[0] + h - 1, X.shape[1] + w - 1))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Y[i: i + h, j: j + w] += X[i, j] * K
    return Y
```

```
def trans_conv(X, K):
    h, w = K.shape
    Y = d2l.zeros((X.shape[0] + h - 1, X.shape[1] + w - 1))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Y[i: i + h, j: j + w] += X[i, j] * K
    return Y
```

17.10.1 基本操作

まずはチャネルを無視して、ストライド 1 でパディングなしの基本的な転置畳み込み操作から始めよう。$n_h \times n_w$ の入力テンソルと、$k_h \times k_w$ のカーネルが与えられたとする。ストライド 1 でカーネルウィンドウを各行で n_w 回、各列で n_h 回動かすと、合計 $n_h n_w$ 個の中間結果が得られる。各中間結果は、ゼロで初期化された $(n_h + k_h - 1) \times (n_w + k_w - 1)$ テンソルである。各中間テンソルを計算するには、入力テンソルの各要素にカーネルを掛け、その結果得られる $k_h \times k_w$ テンソルを各中間テンソルの一部に上書きする。なお、各中間テンソルにおける上書きされる部分の位置は、計算に使われた入力テンソル中の要素の位置に対応している。最後に、これらすべての中間結果を加算して出力を得る。

例として、Fig. 17.10.1 は 2×2 の入力テンソルに対して 2×2 カーネルを用いた転置畳み込みがどのように計算されるかを示している。

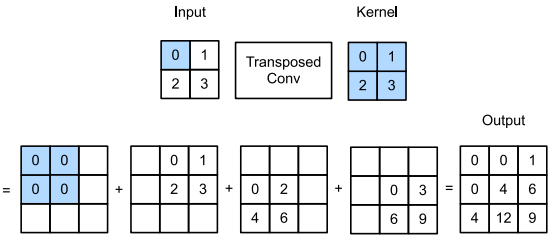


Fig. 17.10.1 2×2 カーネルを用いた転置畳み込み。網掛け部分は、中間テンソルの一部であると同時に、計算に使われた入力テンソルおよびカーネルテンソルの要素を表している。

入力行列 `X` とカーネル行列 `K` に対して、基本的な転置畳み込み操作 `trans_conv` を実装できる。

```
def trans_conv(X, K):
    h, w = K.shape
    Y = d2l.zeros((X.shape[0] + h - 1, X.shape[1] + w - 1))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Y[i: i + h, j: j + w] += X[i, j] * K
    return Y
```

```
def trans_conv(X, K):
    h, w = K.shape
    Y = d2l.zeros((X.shape[0] + h - 1, X.shape[1] + w - 1))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Y[i: i + h, j: j + w] += X[i, j] * K
    return Y
```

```
X = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
```

(continues on next page)

(continued from previous page)

```
K = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
trans_conv(X, K)
```

```
X = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
K = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
trans_conv(X, K)
```

通常の畳み込み（Section 10.2）がカーネルを通して入力要素を縮約するのに対し、転置畳み込みはカーネルを通して入力要素を拡張し、その結果、入力よりも大きい出力を生成する。Fig. 17.10.1 の入力テンソル `X` とカーネルテンソル `K` を構成して、上の基本的な 2 次元転置畳み込み操作の出力を検証できる。

```
X = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
K = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
trans_conv(X, K)
```

```
tensor([[ 0.,  0.,  1.],
        [ 0.,  4.,  6.],
        [ 4., 12.,  9.]])
```

```
X = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
K = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
trans_conv(X, K)
```

```
[07:06:08] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
array([[ 0.,  0.,  1.],
       [ 0.,  4.,  6.],
       [ 4., 12.,  9.]])
```

```
X = d2l.arange(9.0).reshape(3, 3)
K = d2l.tensor([[1.0, 2.0], [3.0, 4.0]])
Y = d2l.corr2d(X, K)
Y
```

```
X = d2l.arange(9.0).reshape(3, 3)
K = d2l.tensor([[1.0, 2.0], [3.0, 4.0]])
Y = d2l.corr2d(X, K)
Y
```

あるいは、入力 `X` とカーネル `K` がどちらも 4 次元テンソルである場合には、高水準 API を使って同じ結果を得ることもできる。

```
X, K = X.reshape(1, 1, 2, 2), K.reshape(1, 1, 2, 2)
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[ 0.,  0.,  1.],
          [ 0.,  4.,  6.],
          [ 4., 12.,  9.]]]], grad_fn=<ConvolutionBackward0>)
```

```
X, K = X.reshape(1, 1, 2, 2), K.reshape(1, 1, 2, 2)
tconv = nn.Conv2DTranspose(1, kernel_size=2)
tconv.initialize(init.Constant(K))
tconv(X)
```

```
array([[[[ 0.,  0.,  1.],
         [ 0.,  4.,  6.],
         [ 4., 12.,  9.]]]])
```

```
def kernel2matrix(K):
    k, W = d2l.zeros(5), d2l.zeros((4, 9))
    k[:2], k[3:5] = K[0, :], K[1, :]
    W[0, :5], W[1, 1:6], W[2, 3:8], W[3, 4:] = k, k, k, k
    return W

W = kernel2matrix(K)
W
```

```
def kernel2matrix(K):
    k, W = d2l.zeros(5), d2l.zeros((4, 9))
    k[:2], k[3:5] = K[0, :], K[1, :]
    W[0, :5], W[1, 1:6], W[2, 3:8], W[3, 4:] = k, k, k, k
    return W

W = kernel2matrix(K)
W
```

17.10.2 パディング、ストライド、複数チャネル

通常の畳み込みではパディングは入力に適用されるが、転置畳み込みでは出力に適用される。たとえば、高さと幅のそれぞれの側におけるパディング数を 1 に指定すると、転置畳み込みの出力から最初と最後の行および列が削除される。

```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, padding=1, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[4.]]]], grad_fn=<ConvolutionBackward0>)
```

```
tconv = nn.Conv2DTranspose(1, kernel_size=2, padding=1)
tconv.initialize(init.Constant(K))
tconv(X)
```

```
array([[[[4.]]]])
```

```
Y == d2l.matmul(W, d2l.reshape(X, -1)).reshape(2, 2)
```

```
Y == d2l.matmul(W, d2l.reshape(X, -1)).reshape(2, 2)
```

転置畳み込みでは、ストライドは入力ではなく中間結果（したがって出力）に対して指定される。Fig. 17.10.1 と同じ入力テンソルおよびカーネルテンソルを用いて、ストライドを 1 から 2 に変更すると、中間テンソルの高さと幅の両方が増加し、その結果、Fig. 17.10.2 の出力テンソルが得られる。

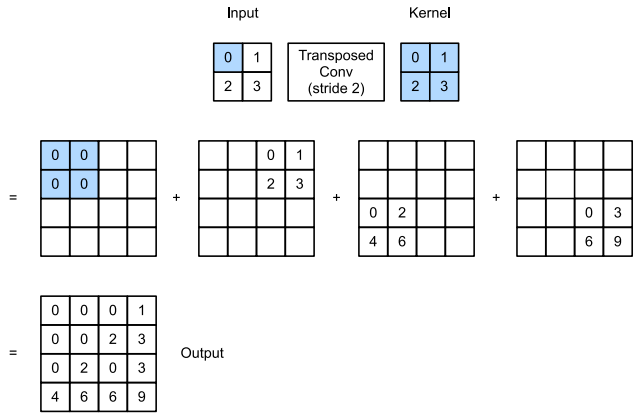


Fig. 17.10.2 ストライド 2 の 2×2 カーネルを用いた転置畳み込み。網掛け部分は、中間テンソルの一部であると同時に、計算に使われた入力テンソルおよびカーネルテンソルの要素を表している。

次のコード片で、Fig. 17.10.2 におけるストライド 2 の転置畳み込み出力を検証できる。

```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, stride=2, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[0., 0., 0., 1.],
          [0., 0., 2., 3.],
```

(continues on next page)

(continued from previous page)

```
          [0., 2., 0., 3.],
          [4., 6., 6., 9.]]]], grad_fn=<ConvolutionBackward0>)
```

```
tconv = nn.Conv2DTranspose(1, kernel_size=2, strides=2)
tconv.initialize(init.Constant(K))
tconv(X)
```

```
array([[[[0., 0., 0., 1.],
         [0., 0., 2., 3.],
         [0., 2., 0., 3.],
         [4., 6., 6., 9.]]]])
```

```
Z = trans_conv(Y, K)
Z == d2l.matmul(W.T, d2l.reshape(Y, -1)).reshape(3, 3)
```

```
Z = trans_conv(Y, K)
Z == d2l.matmul(W.T, d2l.reshape(Y, -1)).reshape(3, 3)
```

複数の入力チャネルと出力チャネルに対しては、転置畳み込みは通常の畳み込みと同じように動作する。入力が c_i チャネルを持つとし、転置畳み込みが各入力チャネルに対して $k_h \times k_w$ のカーネルテンソルを割り当てるとする。複数の出力チャネルが指定される場合、各出力チャネルごとに $c_i \times k_h \times k_w$ のカーネルを持つことになる。

全体として、X を畳み込み層 f に入力して Y = f(X) を出力し、g を f と同じハイパーパラメータを持つ転置畳み込み層として、ただし出力チャネル数だけは X のチャネル数と同じにすると、$g(Y)$ は X と同じ形状になる。これは次の例で示せる。

```
X = torch.rand(size=(1, 10, 16, 16))
conv = nn.Conv2d(10, 20, kernel_size=5, padding=2, stride=3)
tconv = nn.ConvTranspose2d(20, 10, kernel_size=5, padding=2, stride=3)
tconv(conv(X)).shape == X.shape
```

```
True
```

```
X = np.random.uniform(size=(1, 10, 16, 16))
conv = nn.Conv2D(20, kernel_size=5, padding=2, strides=3)
tconv = nn.Conv2DTranspose(10, kernel_size=5, padding=2, strides=3)
conv.initialize()
tconv.initialize()
tconv(conv(X)).shape == X.shape
```

```
True
```

17.10.3 行列の転置との関係

転置畳み込みという名前は、行列の転置に由来する。説明のために、まずは行列積を用いて畳み込みを実装する方法を見てみよう。以下の例では、3×3 の入力 `X` と 2×2 の畳み込みカーネル `K` を定義し、その後 `corr2d` 関数を使って畳み込み出力 `Y` を計算する。

```
X = d2l.arange(9.0).reshape(3, 3)
K = d2l.tensor([[1.0, 2.0], [3.0, 4.0]])
Y = d2l.corr2d(X, K)
Y
```

```
tensor([[27., 37.],
        [57., 67.]])
```

次に、畳み込みカーネル `K` を多くのゼロを含む疎な重み行列 `W` として書き直す。重み行列の形状は (4, 9) で、非ゼロ要素は畳み込みカーネル `K` から来ている。

```
def kernel2matrix(K):
    k, W = d2l.zeros(5), d2l.zeros((4, 9))
    k[:2], k[3:5] = K[0, :], K[1, :]
    W[0, :5], W[1, 1:6], W[2, 3:8], W[3, 4:] = k, k, k, k
    return W

W = kernel2matrix(K)
W
```

```
tensor([[1., 2., 0., 3., 4., 0., 0., 0., 0.],
        [0., 1., 2., 0., 3., 4., 0., 0., 0.],
        [0., 0., 0., 1., 2., 0., 3., 4., 0.],
        [0., 0., 0., 0., 1., 2., 0., 3., 4.]])
```

入力 `X` を行ごとに連結して長さ 9 のベクトルを得る。すると、`W` とベクトル化した `X` の行列積により、長さ 4 のベクトルが得られる。それを再び reshape すると、上の元の畳み込み操作と同じ結果 `Y` が得られる。つまり、畳み込みを行列積で実装したことになる。

```
Y == d2l.matmul(W, d2l.reshape(X, -1)).reshape(2, 2)
```

```
tensor([[True, True],
        [True, True]])
```

同様に、転置畳み込みも行列積を用いて実装できる。次の例では、上の通常の

畳み込みから得られた 2×2 の出力 `Y` を転置畳み込みの入力として用いる。この操作を行列積で実装するには、重み行列 `W` を転置して新しい形状 $(9, 4)$ にすればよいだけである。

```
Z = trans_conv(Y, K)
Z == d2l.matmul(W.T, d2l.reshape(Y, -1)).reshape(3, 3)
```

```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

行列積による畳み込みの実装を考えよう。入力ベクトル $\mathbf{x}$ と重み行列 $\mathbf{W}$ が与えられたとき、畳み込みの順伝播関数は、入力に重み行列を掛けて出力ベクトル $\mathbf{y} = \mathbf{W}\mathbf{x}$ を返すことで実装できる。逆伝播は連鎖律に従い、$\nabla_{\mathbf{x}}\mathbf{y} = \mathbf{W}^\top$ であるため、畳み込みの逆伝播関数は入力に転置された重み行列 $\mathbf{W}^\top$ を掛けることで実装できる。したがって、転置畳み込み層は畳み込み層の順伝播関数と逆伝播関数を入れ替えたものとみなせる。つまり、その順伝播関数と逆伝播関数は、それぞれ入力ベクトルに $\mathbf{W}^\top$ と $\mathbf{W}$ を掛ける。

17.10.4 まとめ

- 通常の畳み込みがカーネルを通して入力要素を縮約するのに対し、転置畳み込みはカーネルを通して入力要素を拡張し、その結果、入力よりも大きい出力を生成する。
- X を畳み込み層 f に入力して $\mathsf{Y} = f(\mathsf{X})$ を出力し、g を f と同じハイパーパラメータを持つ転置畳み込み層として、ただし出力チャネル数だけは X のチャネル数と同じにすると、$g(Y)$ は X と同じ形状になる。
- 畳み込みは行列積を用いて実装できる。転置畳み込み層は、畳み込み層の順伝播関数と逆伝播関数を入れ替えたものとみなせる。

17.10.5 演習

1. Section 17.10.3 では、畳み込みの入力 `X` と転置畳み込みの出力 `Z` は同じ形状を持つ。では、それらは同じ値だろうか。なぜか。
2. 畳み込みを実装するのに行列積を使うのは効率的だろうか。なぜか。

17.11 完全畳み込みネットワーク

Section 17.9 で述べたように、セマンティックセグメンテーションは画像をピクセル単位で分類する。完全畳み込みネットワーク（FCN; fully convolutional network）は、畳み込みニューラルネットワークを用いて画像の各ピクセルをピ

クセルクラスへ変換する (Long *et al.*, 2015)。これまで画像分類や物体検出で見てきた CNN とは異なり、完全畳み込みネットワークは中間特徴マップの高さと幅を入力画像のそれに戻する。これは Section 17.10 で導入した転置畳み込み層によって実現される。その結果、分類出力と入力画像はピクセルレベルで 1 対 1 に対応する。すなわち、任意の出力ピクセルにおけるチャネル次元には、同じ空間位置にある入力ピクセルに対する分類結果が格納される。

```
%matplotlib inline
from d2l import torch as d2l
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()
```

```
batch_size, crop_size = 32, (320, 480)
train_iter, test_iter = d2l.load_data_voc(batch_size, crop_size)
```

```
batch_size, crop_size = 32, (320, 480)
train_iter, test_iter = d2l.load_data_voc(batch_size, crop_size)
```

17.11.1 モデル

ここでは、完全畳み込みネットワークモデルの基本設計を説明する。Fig. 17.11.1 に示すように、このモデルはまず CNN を用いて画像特徴を抽出し、次に 1×1 畳み込み層を介してチャネル数をクラス数へ変換し、最後に Section 17.10 で導入した転置畳み込みを用いて特徴マップの高さと幅を入力画像のそれへ変換する。その結果、モデル出力は入力画像と同じ高さと幅を持ち、出力チャネルには同じ空間位置にある入力ピクセルの予測クラスが含まれる。

以下では、ImageNet データセットで事前学習された ResNet-18 モデルを用いて画像特徴を抽出し、そのモデルインスタンスを `pretrained_net` と表す。このモデルの最後の数層にはグローバル平均プーリング層と全結合層が含まれているが、これらは完全畳み込みネットワークでは不要である。

```
pretrained_net = torchvision.models.resnet18(pretrained=True)
list(pretrained_net.children())[-3:]
```

```
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /
```

(continues on next page)

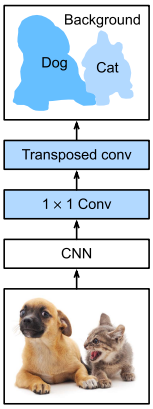


Fig. 17.11.1 完全畳み込みネットワーク。

(continued from previous page)

```
↪home/ci/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
100%|          | 44.7M/44.7M [00:00<00:00, 59.3MB/s]
```

```
[Sequential(
   (0): BasicBlock(
     (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,␣
↪1), bias=False)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
     (relu): ReLU(inplace=True)
     (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,␣
↪1), bias=False)
     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
     (downsample): Sequential(
       (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
       (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
     )
   )
   (1): BasicBlock(
     (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,␣
↪1), bias=False)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
     (relu): ReLU(inplace=True)
     (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,␣
↪1), bias=False)
     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
   )
 ),
 AdaptiveAvgPool2d(output_size=(1, 1)),
 Linear(in_features=512, out_features=1000, bias=True)]
```

```
pretrained_net = gluon.model_zoo.vision.resnet18_v2(pretrained=True)
pretrained_net.features[-3:], pretrained_net.output
```

```
[07:22:38] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(HybridSequential(
   (0): Activation(relu)
   (1): GlobalAvgPool2D(size=(1, 1), stride=(1, 1), padding=(0, 0), ceil_
↪mode=True, global_pool=True, pool_type=avg, layout=NCHW)
   (2): Flatten
 ),
 Dense(512 -> 1000, linear))
```

次に、完全畳み込みネットワークのインスタンス net を作成する。これは、出力に最も近い最後のグローバル平均プーリング層と全結合層を除く、ResNet-18 の事前学習済み層をすべてコピーする。

```
net = nn.Sequential(*list(pretrained_net.children())[:-2])
```

```
net = nn.HybridSequential()
for layer in pretrained_net.features[:-2]:
    net.add(layer)
```

高さと幅がそれぞれ 320 と 480 の入力を与えると、net の順伝播は入力の高さと幅を元の 1/32、すなわち 10 と 15 にまで縮小する。

```
X = torch.rand(size=(1, 3, 320, 480))
net(X).shape
```

```
torch.Size([1, 512, 10, 15])
```

```
X = np.random.uniform(size=(1, 3, 320, 480))
net(X).shape
```

```
(1, 512, 10, 15)
```

次に、1×1 畳み込み層を用いて出力チャネル数を Pascal VOC2012 データセットのクラス数（21）へ変換する。最後に、特徴マップの高さと幅を **32 倍に増やして**入力画像の高さと幅に戻す必要がある。Section 10.3 で畳み込み層の出力形状の計算方法を思い出してほしい。$(320 - 64 + 16 \times 2 + 32)/32 = 10$ かつ $(480 - 64 + 16 \times 2 + 32)/32 = 15$ なので、ストライドを 32 に設定した転置畳み込み層を構成し、カーネルの高さと幅を 64、パディングを 16 に設定する。一般に、

ストライドを s、パディングを $s/2$（$s/2$ が整数であると仮定）、カーネルの高さと幅を $2s$ とすると、転置畳み込みは入力の高さと幅を s 倍に増やする。

```
num_classes = 21
net.add_module('final_conv', nn.Conv2d(512, num_classes, kernel_size=1))
net.add_module('transpose_conv', nn.ConvTranspose2d(num_classes, num_classes,
                                    kernel_size=64, padding=16, stride=32))
```

```
num_classes = 21
net.add(nn.Conv2D(num_classes, kernel_size=1),
        nn.Conv2DTranspose(
            num_classes, kernel_size=64, padding=16, strides=32))
```

17.11.2 転置畳み込み層の初期化

すでに見たように、転置畳み込み層は特徴マップの高さと幅を増やすことができる。画像処理では、画像を拡大する、すなわち アップサンプリングが必要になることがある。バイリニア補間は、よく使われるアップサンプリング手法の 1 つである。これは転置畳み込み層の初期化にもよく用いられる。

バイリニア補間を説明するために、入力画像が与えられたときにアップサンプリング後の出力画像の各ピクセルを計算したいとする。出力画像の座標 (x, y) にあるピクセルを計算するには、まず (x, y) を入力画像上の座標 (x', y') に写像する。たとえば、入力サイズと出力サイズの比に従いる。写像された x' と y' は実数であることに注意しよ。次に、入力画像上で座標 (x', y') に最も近い 4 つのピクセルを見つける。最後に、座標 (x, y) にある出力画像のピクセルは、入力画像上のこの 4 つの最近傍ピクセルと、(x', y') からの相対距離に基づいて計算される。

バイリニア補間によるアップサンプリングは、以下の `bilinear_kernel` 関数で構成したカーネルを持つ転置畳み込み層によって実装できる。紙幅の都合上、ここでは `bilinear_kernel` 関数の実装のみを示し、そのアルゴリズム設計についての議論は省略する。

```
def bilinear_kernel(in_channels, out_channels, kernel_size):
    factor = (kernel_size + 1) // 2
    if kernel_size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = (torch.arange(kernel_size).reshape(-1, 1),
          torch.arange(kernel_size).reshape(1, -1))
    filt = (1 - torch.abs(og[0] - center) / factor) * \
           (1 - torch.abs(og[1] - center) / factor)
    weight = torch.zeros((in_channels, out_channels,
                          kernel_size, kernel_size))
    weight[range(in_channels), range(out_channels), :, :] = filt
    return weight
```

```
def bilinear_kernel(in_channels, out_channels, kernel_size):
    factor = (kernel_size + 1) // 2
    if kernel_size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = (np.arange(kernel_size).reshape(-1, 1),
          np.arange(kernel_size).reshape(1, -1))
    filt = (1 - np.abs(og[0] - center) / factor) * \
           (1 - np.abs(og[1] - center) / factor)
    weight = np.zeros((in_channels, out_channels, kernel_size, kernel_size))
    weight[range(in_channels), range(out_channels), :, :] = filt
    return np.array(weight)
```

転置畳み込み層で実装されたバイリニア補間によるアップサンプリングを試してみましょう。高さと幅を 2 倍にする転置畳み込み層を構成し、そのカーネルを `bilinear_kernel` 関数で初期化する。

```
conv_trans = nn.ConvTranspose2d(3, 3, kernel_size=4, padding=1, stride=2,
                                bias=False)
conv_trans.weight.data.copy_(bilinear_kernel(3, 3, 4));
```

```
conv_trans = nn.Conv2DTranspose(3, kernel_size=4, padding=1, strides=2)
conv_trans.initialize(init.Constant(bilinear_kernel(3, 3, 4)))
```

画像 `X` を読み込み、アップサンプリングの出力を `Y` に代入する。画像を表示するために、チャネル次元の位置を調整する必要がある。

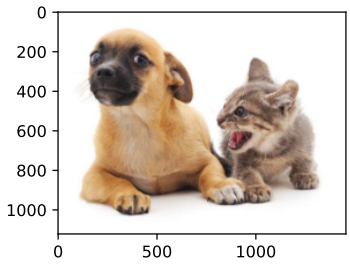
```
img = torchvision.transforms.ToTensor()(d2l.Image.open('../img/catdog.jpg'))
X = img.unsqueeze(0)
Y = conv_trans(X)
out_img = Y[0].permute(1, 2, 0).detach()
```

```
img = image.imread('../img/catdog.jpg')
X = np.expand_dims(img.astype('float32').transpose(2, 0, 1), axis=0) / 255
Y = conv_trans(X)
out_img = Y[0].transpose(1, 2, 0)
```

ご覧のとおり、転置畳み込み層は画像の高さと幅の両方を 2 倍にする。座標のスケールが異なることを除けば、バイリニア補間で拡大した画像と Section 17.3 で表示した元の画像は同じように見える。

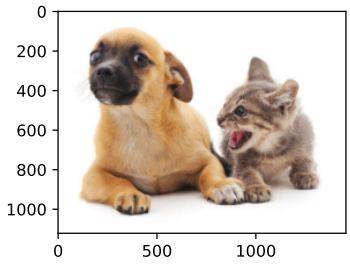
```
d2l.set_figsize()
print('input image shape:', img.permute(1, 2, 0).shape)
d2l.plt.imshow(img.permute(1, 2, 0));
print('output image shape:', out_img.shape)
d2l.plt.imshow(out_img);
```

```
input image shape: torch.Size([561, 728, 3])
output image shape: torch.Size([1122, 1456, 3])
```




```
d2l.set_figsize()
print('input image shape:', img.shape)
d2l.plt.imshow(img.asnumpy());
print('output image shape:', out_img.shape)
d2l.plt.imshow(out_img.asnumpy());
```

```
input image shape: (561, 728, 3)
output image shape: (1122, 1456, 3)
```



完全畳み込みネットワークでは、転置畳み込み層をバイリニア補間によるアップサンプリングで初期化する。1×1 畳み込み層には Xavier 初期化を用いる。

```
W = bilinear_kernel(num_classes, num_classes, 64)
net.transpose_conv.weight.data.copy_(W);
```

```
W = bilinear_kernel(num_classes, num_classes, 64)
net[-1].initialize(init.Constant(W))
net[-2].initialize(init=init.Xavier())
```

17.11.3 データセットの読み込み

Section 17.9 で導入したセマンティックセグメンテーションデータセットを読み込みる。ランダムクロップの出力画像形状は 320×480 に指定する。高さと幅の両方が 32 で割り切れるようにする。

```
batch_size, crop_size = 32, (320, 480)
train_iter, test_iter = d2l.load_data_voc(batch_size, crop_size)
```

```
Downloading ../data/VOCtrainval_11-May-2012.tar from http://d2l-data.s3-
↪accelerate.amazonaws.com/VOCtrainval_11-May-2012.tar...
read 1114 examples
read 1078 examples
```

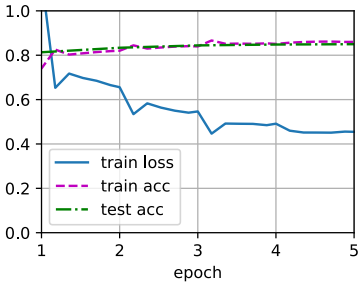
17.11.4 学習

これで、構築した完全畳み込みネットワークを学習できる。ここでの損失関数と精度計算は、以前の章の画像分類と本質的には変わらない。転置畳み込み層の出力チャネルを用いて各ピクセルのクラスを予測するため、損失計算ではチャネル次元が指定される。さらに、精度はすべてのピクセルについての予測クラスの正しさに基づいて計算される。

```
def loss(inputs, targets):
    return F.cross_entropy(inputs, targets, reduction='none').mean(1).mean(1)

num_epochs, lr, wd, devices = 5, 0.001, 1e-3, d2l.try_all_gpus()
trainer = torch.optim.SGD(net.parameters(), lr=lr, weight_decay=wd)
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.455, train acc 0.860, test acc 0.850
258.7 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



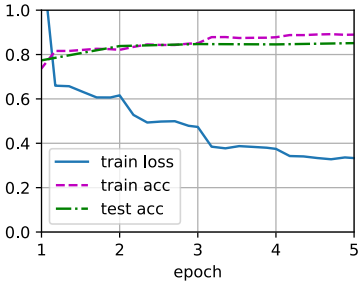

```
num_epochs, lr, wd, devices = 5, 0.1, 1e-3, d2l.try_all_gpus()
loss = gluon.loss.SoftmaxCrossEntropyLoss(axis=1)
net.collect_params().reset_ctx(devices)
trainer = gluon.Trainer(net.collect_params(), 'sgd',
```

(continues on next page)

(continued from previous page)

```
                    {'learning_rate': lr, 'wd': wd})
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.333, train acc 0.890, test acc 0.851
129.1 examples/sec on [gpu(0), gpu(1)]
```



17.11.5 予測

予測時には、入力画像の各チャネルを標準化し、画像を CNN が必要とする 4 次元の入力形式に変換する必要がある。

```
def predict(img):
    X = test_iter.dataset.normalize_image(img).unsqueeze(0)
    pred = net(X.to(devices[0])).argmax(dim=1)
    return pred.reshape(pred.shape[1], pred.shape[2])
```

```
def predict(img):
    X = test_iter._dataset.normalize_image(img)
    X = np.expand_dims(X.transpose(2, 0, 1), axis=0)
    pred = net(X.as_in_ctx(devices[0])).argmax(axis=1)
    return pred.reshape(pred.shape[1], pred.shape[2])
```

各ピクセルの予測クラスを可視化するために、予測されたクラスをデータセット内のラベル色に戻する。

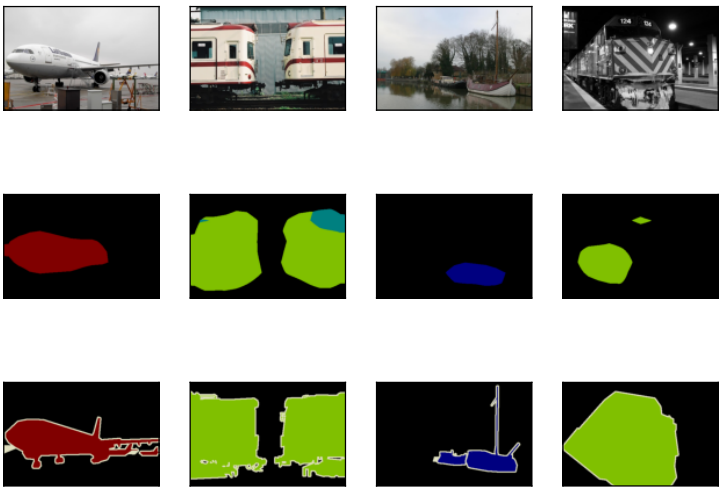
```
def label2image(pred):
    colormap = torch.tensor(d2l.VOC_COLORMAP, device=devices[0])
    X = pred.long()
    return colormap[X, :]
```

```
def label2image(pred):
    colormap = np.array(d2l.VOC_COLORMAP, ctx=devices[0], dtype='uint8')
    X = pred.astype('int32')
    return colormap[X, :]
```

テストデータセットの画像はサイズも形状もさまざまである。モデルはストライド 32 の転置畳み込み層を使うため、入力画像の高さまたは幅が 32 で割り切れない場合、転置畳み込み層の出力の高さまたは幅は入力画像の形状とずれる。この問題に対処するために、画像内で高さと幅が 32 の整数倍である複数の矩形領域を切り出し、それぞれの領域の画素に対して個別に順伝播を行うことができる。なお、これらの矩形領域の和集合が入力画像全体を完全に覆う必要がある。あるピクセルが複数の矩形領域に含まれる場合、同じピクセルに対する別々の領域での転置畳み込み出力の平均を softmax 演算への入力としてクラスを予測できる。

簡単のため、ここでは大きめのテスト画像をいくつか読み込み、画像の左上隅から始まる 320×480 の領域を予測に用いる。これらのテスト画像について、切り出した領域、予測結果、および正解を行ごとに表示する。

```
voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
test_images, test_labels = d2l.read_voc_images(voc_dir, False)
n, imgs = 4, []
for i in range(n):
    crop_rect = (0, 0, 320, 480)
    X = torchvision.transforms.functional.crop(test_images[i], *crop_rect)
    pred = label2image(predict(X))
    imgs += [X.permute(1,2,0), pred.cpu(),
             torchvision.transforms.functional.crop(
                 test_labels[i], *crop_rect).permute(1,2,0)]
d2l.show_images(imgs[::3] + imgs[1::3] + imgs[2::3], 3, n, scale=2);
```

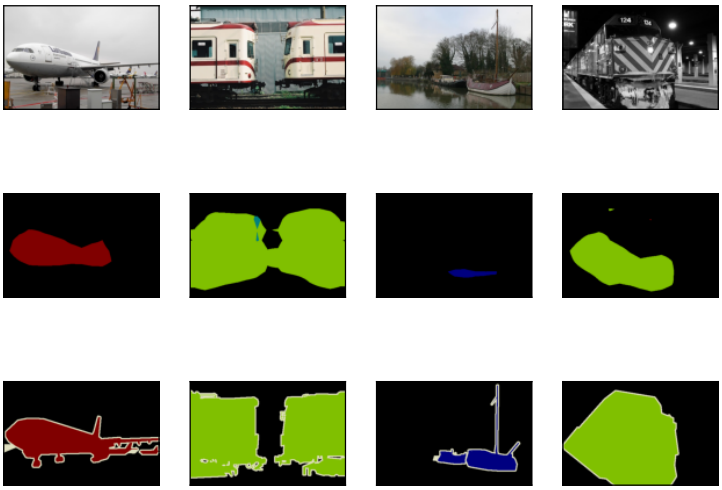


```
voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
test_images, test_labels = d2l.read_voc_images(voc_dir, False)
n, imgs = 4, []
for i in range(n):
    crop_rect = (0, 0, 480, 320)
    X = image.fixed_crop(test_images[i], *crop_rect)
```

(continues on next page)

(continued from previous page)

```
    pred = label2image(predict(X))
    imgs += [X, pred, image.fixed_crop(test_labels[i], *crop_rect)]
d2l.show_images(imgs[::3] + imgs[1::3] + imgs[2::3], 3, n, scale=2);
```



17.11.6 まとめ

- 完全畳み込みネットワークはまず CNN を用いて画像特徴を抽出し、次に 1×1 畳み込み層を介してチャネル数をクラス数へ変換し、最後に転置畳み込みを用いて特徴マップの高さと幅を入力画像のそれへ変換する。
- 完全畳み込みネットワークでは、転置畳み込み層の初期化にバイリニア補間によるアップサンプリングを用いることができる。

17.11.7 演習

1. 実験で転置畳み込み層に Xavier 初期化を用いると、結果はどのように変わりますか？
2. ハイパーパラメータを調整することで、モデルの精度をさらに改善できるか？
3. テスト画像内のすべてのピクセルのクラスを予測しよ。
4. 元の完全畳み込みネットワークの論文では、いくつかの中間 CNN 層の出力も使用している (Long *et al.*, 2015)。このアイデアを実装してみよ。

17.12 ニューラル・スタイル変換

写真好きの方なら、フィルターに馴染みがあるかもしれない。フィルターは写真の色調を変え、風景写真をより鮮明にしたり、ポートレート写真の肌を白く見せたりできる。しかし、通常ひとつのフィルターは写真の一側面しか変えない。写真に理想的なスタイルを適用するには、おそらくさまざまなフィルターの組み合わせを何度も試す必要がある。この過程は、モデルのハイパーパラメータ調整と同じくらい複雑である。

この節では、CNN の層ごとの表現を活用して、ある画像のスタイルを別の画像に自動的に適用する、すなわち スタイル変換 (Gatys *et al.*, 2016) を行う。このタスクには 2 枚の入力画像が必要である。1 枚は コンテンツ画像、もう 1 枚はスタイル画像である。ニューラルネットワークを用いてコンテンツ画像を修正し、スタイル画像のスタイルに近づける。たとえば、Fig. 17.12.1 のコンテンツ画像は、シアトル郊外のマウント・レーニア国立公園で私たちが撮影した風景写真であり、スタイル画像は秋のオークの木々をテーマにした油絵である。出力される合成画像では、スタイル画像の油彩の筆致が適用されてより鮮やかな色彩になりつつ、コンテンツ画像中の物体の主要な形状は保たれる。

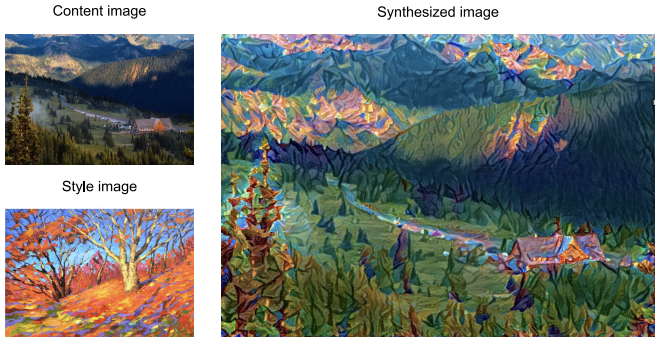


Fig. 17.12.1 コンテンツ画像とスタイル画像が与えられると、スタイル変換は合成画像を出力する。

17.12.1 手法

Fig. 17.12.2 は、簡略化した例を用いて CNN ベースのスタイル変換手法を示している。まず、合成画像を初期化する。たとえばコンテンツ画像で初期化する。この合成画像は、スタイル変換の過程で更新が必要な唯一の変数、すなわち学習中に更新されるモデルパラメータである。次に、画像特徴を抽出するための事前学習済み CNN を選び、学習中はそのモデルパラメータを固定する。この深い CNN は複数の層を用いて画像の階層的特徴を抽出す。これらの層の出力の一部をコンテンツ特徴またはスタイル特徴として選ぶことができる。Fig. 17.12.2 を例にとると、ここでの事前学習済みニューラルネットワークは 3 つの畳み込

み層を持ち、2 層目の出力がコンテンツ特徴、1 層目と 3 層目の出力がスタイル特徴になる。

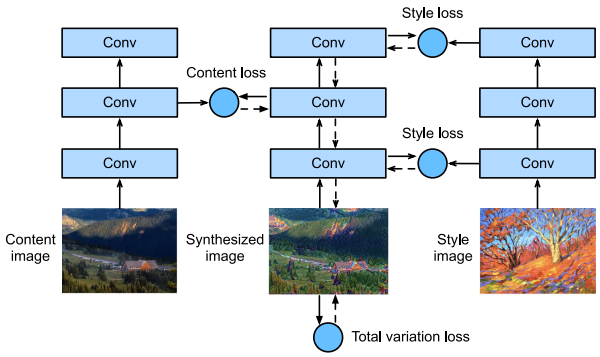


Fig. 17.12.2 CNN ベースのスタイル変換の過程。実線は順伝播の方向を、点線は逆伝播の方向を示す。

次に、順伝播（実線矢印の方向）によってスタイル変換の損失関数を計算し、逆伝播（破線矢印の方向）によってモデルパラメータ（出力される合成画像）を更新する。スタイル変換で一般的に用いられる損失関数は 3 つの部分から成る。(i) コンテンツ損失は、合成画像とコンテンツ画像をコンテンツ特徴の面で近づける。(ii) スタイル損失は、合成画像とスタイル画像をスタイル特徴の面で近づける。(iii) 全変動損失は、合成画像中のノイズを減らすのに役立ちる。最後に、モデルの学習が終わったら、スタイル変換のモデルパラメータを出力して最終的な合成画像を生成する。

以下では、具体的な実験を通してスタイル変換の技術的詳細を説明する。

17.12.2 コンテンツ画像とスタイル画像の読み込み

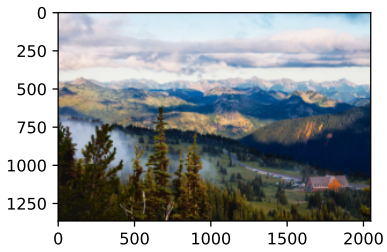
まず、コンテンツ画像とスタイル画像を読み込みる。表示された座標軸から、これらの画像のサイズが異なることがわかる。

```
%matplotlib inline
from d2l import torch as d2l
import torch
import torchvision
from torch import nn

d2l.set_figsize()
content_img = d2l.Image.open('../img/rainier.jpg')
d2l.plt.imshow(content_img);
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, image, init, np, npx
from mxnet.gluon import nn
```

(continues on next page)

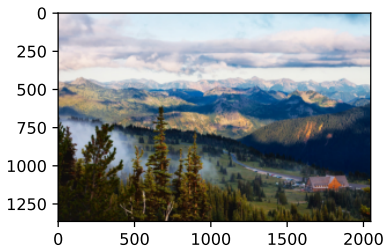


(continued from previous page)

```
npx.set_np()

d2l.set_figsize()
content_img = image.imread('../img/rainier.jpg')
d2l.plt.imshow(content_img.asnumpy());
```

```
[07:38:08] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```




```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

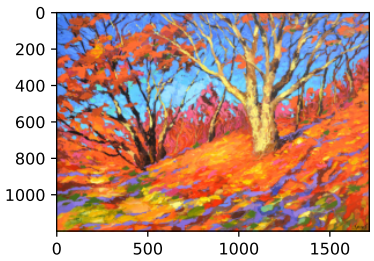
```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

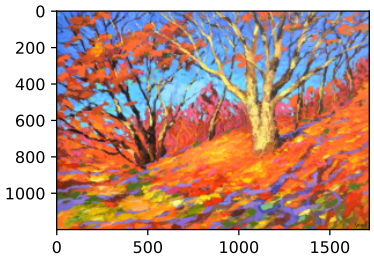
```
style_img = d2l.Image.open('../img/autumn-oak.jpg')
d2l.plt.imshow(style_img);
```

```
style_img = image.imread('../img/autumn-oak.jpg')
d2l.plt.imshow(style_img.asnumpy());
```

```
def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
```

(continues on next page)





(continued from previous page)

```
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

```
def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

17.12.3 前処理と後処理

以下では、画像の前処理と後処理を行う 2 つの関数を定義する。preprocess 関数は、入力画像の 3 つの RGB チャンネルそれぞれを標準化し、その結果を CNN の入力形式に変換する。postprocess 関数は、出力画像の画素値を標準化前の元の値に戻する。画像表示関数は各画素が 0 から 1 の浮動小数点値であることを要求するため、0 未満の値は 0 に、1 を超える値は 1 にそれぞれ置き換える。

```
rgb_mean = torch.tensor([0.485, 0.456, 0.406])
rgb_std = torch.tensor([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
    transforms = torchvision.transforms.Compose([
        torchvision.transforms.Resize(image_shape),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=rgb_mean, std=rgb_std)])
    return transforms(img).unsqueeze(0)

def postprocess(img):
    img = img[0].to(rgb_std.device)
    img = torch.clamp(img.permute(1, 2, 0) * rgb_std + rgb_mean, 0, 1)
    return torchvision.transforms.ToPILImage()(img.permute(2, 0, 1))
```

```
rgb_mean = np.array([0.485, 0.456, 0.406])
rgb_std = np.array([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
    img = image.imresize(img, *image_shape)
    img = (img.astype('float32') / 255 - rgb_mean) / rgb_std
    return np.expand_dims(img.transpose(2, 0, 1), axis=0)

def postprocess(img):
    img = img[0].as_in_ctx(rgb_std.ctx)
    return (img.transpose(1, 2, 0) * rgb_std + rgb_mean).clip(0, 1)
```

```
def gram(X):
    num_channels, n = X.shape[1], d2l.size(X) // X.shape[1]
    X = d2l.reshape(X, (num_channels, n))
    return d2l.matmul(X, X.T) / (num_channels * n)
```

```
def gram(X):
    num_channels, n = X.shape[1], d2l.size(X) // X.shape[1]
    X = d2l.reshape(X, (num_channels, n))
    return d2l.matmul(X, X.T) / (num_channels * n)
```

17.12.4 特徴の抽出

画像特徴を抽出するために、ImageNet データセットで事前学習された VGG-19 モデルを用いる (Gatys *et al.*, 2016)。

```
pretrained_net = torchvision.models.vgg19(pretrained=True)
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /home/
↪ci/.cache/torch/hub/checkpoints/vgg19-dcbb9e9d.pth
100%|          | 548M/548M [00:02<00:00, 227MB/s]
```

```
pretrained_net = gluon.model_zoo.vision.vgg19(pretrained=True)
```

```
Downloading /opt/mxnet/models/vgg19-ad2f660d.zipbf5c08ab-ae7d-450e-b61d-
↪940f6c92e722 from https://apache-mxnet.s3-accelerate.dualstack.amazonaws.com/
↪gluon/models/vgg19-ad2f660d.zip...
```

```
def tv_loss(Y_hat):
    return 0.5 * (d2l.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  d2l.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

```
def tv_loss(Y_hat):
    return 0.5 * (d2l.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  d2l.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

画像のコンテンツ特徴とスタイル特徴を抽出するために、VGG ネットワークの特定の層の出力を選ぶことができる。一般に、入力層に近いほど画像の細部を抽出しやすく、逆に出力層に近いほど画像の全体的な情報を抽出しやすくなる。合成画像にコンテンツ画像の細部を過度に残さないようにするため、出力層に近い VGG 層を コンテンツ層として選び、画像のコンテンツ特徴を出力させる。また、局所的および大域的なスタイル特徴を抽出するために、異なる VGG 層の出力を選ぶ。これらの層は スタイル層とも呼ばれる。Section 11.2 で述べたように、VGG ネットワークは 5 つの畳み込みブロックを使う。この実験では、第 4 畳み込みブロックの最後の畳み込み層をコンテンツ層として選び、各畳み込みブロックの最初の畳み込み層をスタイル層として選ぶ。これらの層のインデックスは pretrained_net インスタンスを表示することで得られる。

```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

```
content_weight, style_weight, tv_weight = 1, 1e4, 10

def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # Calculate the content, style, and total variance losses respectively
    contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
        contents_Y_hat, contents_Y)]
    styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
        styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # Add up all the losses
    l = sum(styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

```
content_weight, style_weight, tv_weight = 1, 1e4, 10
```

(continues on next page)

(continued from previous page)

```
def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # Calculate the content, style, and total variance losses respectively
    contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
        contents_Y_hat, contents_Y)]
    styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
        styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # Add up all the losses
    l = sum(styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

VGG 層を用いて特徴を抽出する際には、入力層から、出力層に最も近いコンテンツ層またはスタイル層までの層だけを使えば十分である。特徴抽出に使う VGG 層だけを残した新しいネットワークインスタンス net を構築しよう。

```
net = nn.Sequential(*[pretrained_net.features[i] for i in
                      range(max(content_layers + style_layers) + 1)])
```

```
net = nn.Sequential()
for i in range(max(content_layers + style_layers) + 1):
    net.add(pretrained_net.features[i])
```

入力 X が与えられたとき、単に順伝播 net(X) を呼び出すだけでは最後の層の出力しか得られない。中間層の出力も必要なので、層ごとに計算を行い、コンテンツ層とスタイル層の出力を保持する必要がある。

```
def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

以下に 2 つの関数を定義する。get_contents 関数はコンテンツ画像からコンテンツ特徴を抽出し、get_styles 関数はスタイル画像からスタイル特徴を抽出す。学習中に事前学習済み VGG のモデルパラメータを更新する必要はないため、学習開始前にコンテンツ特徴とスタイル特徴を抽出しておくことができる。合成画像はスタイル変換のために更新されるモデルパラメータの集合なので、学習中に extract_features 関数を呼び出すことでのみ、合成画像のコンテンツ特徴とスタイル特徴を抽出できる。

```
def get_contents(image_shape, device):
```

(continues on next page)

(continued from previous page)

```
    content_X = preprocess(content_img, image_shape).to(device)
    contents_Y, _ = extract_features(content_X, content_layers, style_layers)
    return content_X, contents_Y

def get_styles(image_shape, device):
    style_X = preprocess(style_img, image_shape).to(device)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

```
def get_contents(image_shape, device):
    content_X = preprocess(content_img, image_shape).copyto(device)
    contents_Y, _ = extract_features(content_X, content_layers, style_layers)
    return content_X, contents_Y

def get_styles(image_shape, device):
    style_X = preprocess(style_img, image_shape).copyto(device)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

17.12.5 損失関数の定義

ここでは、スタイル変換の損失関数を説明する。損失関数はコンテンツ損失、スタイル損失、全変動損失から成る。

コンテンツ損失

線形回帰の損失関数と同様に、コンテンツ損失は二乗損失関数を通じて合成画像とコンテンツ画像の間のコンテンツ特徴の違いを測りる。二乗損失関数の 2 つの入力は、どちらも extract_features 関数で計算されたコンテンツ層の出力である。

```
def content_loss(Y_hat, Y):
    # We detach the target content from the tree used to dynamically compute
    # the gradient: this is a stated value, not a variable. Otherwise the loss
    # will throw an error.
    return torch.square(Y_hat - Y.detach()).mean()
```

```
def content_loss(Y_hat, Y):
    return np.square(Y_hat - Y).mean()
```

スタイル損失

スタイル損失もコンテンツ損失と同様に、二乗損失関数を用いて合成画像とスタイル画像のスタイルの違いを測りる。任意のスタイル層のスタイル出力を表すために、まず extract_features 関数を使ってスタイル層の出力を計算する。出力が 1 つのサンプル、c チャンネル、高さ h、幅 w を持つとすると、この出力

を c 行 hw 列の行列 $\mathbf{X}$ に変換できる。この行列は、長さ hw の c 個のベクトル $\mathbf{x}_1, \ldots, \mathbf{x}_c$ の連結とみなせる。ここで、ベクトル $\mathbf{x}_i$ はチャンネル i のスタイル特徴を表す。

これらのベクトル $\mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{c \times c}$ の グラム行列では、i 行 j 列の要素 x_{ij} はベクトル $\mathbf{x}_i$ と $\mathbf{x}_j$ の内積である。これはチャンネル i と j のスタイル特徴の相関を表す。このグラム行列を任意のスタイル層のスタイル出力として用いる。hw の値が大きいほど、グラム行列の値も大きくなりやすいことに注意しよ。また、グラム行列の縦横のサイズはどちらもチャンネル数 c である。スタイル損失がこれらの値の影響を受けないようにするため、以下の gram 関数ではグラム行列をその要素数、すなわち chw で割っている。

```
def gram(X):
    num_channels, n = X.shape[1], d2l.size(X) // X.shape[1]
    X = d2l.reshape(X, (num_channels, n))
    return d2l.matmul(X, X.T) / (num_channels * n)
```

明らかに、スタイル損失の二乗損失関数における 2 つのグラム行列入力は、合成画像とスタイル画像のスタイル層出力に基づいている。ここでは、スタイル画像に基づくグラム行列 gram_Y は事前に計算済みであると仮定する。

```
def style_loss(Y_hat, gram_Y):
    return torch.square(gram(Y_hat) - gram_Y.detach()).mean()
```

```
def style_loss(Y_hat, gram_Y):
    return np.square(gram(Y_hat) - gram_Y).mean()
```

全変動損失

ときには、学習された合成画像に高周波ノイズ、すなわち特に明るい画素や暗い画素が多く含まれることがある。一般的なノイズ低減法の 1 つは 全変動ノイズ除去である。座標 (i, j) における画素値を $x_{i,j}$ とする。全変動損失を減らすと

$$\sum_{i,j} \left|x_{i,j} - x_{i+1,j}\right| + \left|x_{i,j} - x_{i,j+1}\right| \tag{17.12.1}$$

合成画像上の隣接画素の値がより近くなる。

```
def tv_loss(Y_hat):
    return 0.5 * (d2l.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  d2l.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

損失関数

スタイル変換の損失関数は、コンテンツ損失、スタイル損失、全変動損失の重み付き和です。これらの重みハイパーパラメータを調整することで、合成画像

におけるコンテンツ保持、スタイル変換、ノイズ低減のバランスを取ることができる。

```
content_weight, style_weight, tv_weight = 1, 1e4, 10

def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # Calculate the content, style, and total variance losses respectively
    contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
        contents_Y_hat, contents_Y)]
    styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
        styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # Add up all the losses
    l = sum(styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

17.12.6 合成画像の初期化

スタイル変換では、合成画像は学習中に更新が必要な唯一の変数である。したがって、単純なモデル SynthesizedImage を定義し、合成画像をモデルパラメータとして扱うことができる。このモデルでは、順伝播は単にモデルパラメータを返すだけである。

```
class SynthesizedImage(nn.Module):
    def __init__(self, img_shape, **kwargs):
        super(SynthesizedImage, self).__init__(**kwargs)
        self.weight = nn.Parameter(torch.rand(*img_shape))

    def forward(self):
        return self.weight
```

```
class SynthesizedImage(nn.Block):
    def __init__(self, img_shape, **kwargs):
        super(SynthesizedImage, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=img_shape)

    def forward(self):
        return self.weight.data()
```

次に、get_inits 関数を定義する。この関数は合成画像モデルのインスタンスを作成し、それを画像 X で初期化する。さまざまなスタイル層におけるスタイル画像のグラム行列 styles_Y_gram は、学習前に計算される。

```
def get_inits(X, device, lr, styles_Y):
    gen_img = SynthesizedImage(X.shape).to(device)
    gen_img.weight.data.copy_(X.data)
    trainer = torch.optim.Adam(gen_img.parameters(), lr=lr)
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

```
def get_inits(X, device, lr, styles_Y):
    gen_img = SynthesizedImage(X.shape)
    gen_img.initialize(init.Constant(X), ctx=device, force_reinit=True)
    trainer = gluon.Trainer(gen_img.collect_params(), 'adam',
                            {'learning_rate': lr})
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

17.12.7 学習

スタイル変換のモデルを学習するとき、合成画像のコンテンツ特徴とスタイル特徴を継続的に抽出し、損失関数を計算する。以下に学習ループを定義する。

```
def train(X, contents_Y, styles_Y, device, lr, num_epochs, lr_decay_epoch):
    X, styles_Y_gram, trainer = get_inits(X, device, lr, styles_Y)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_decay_epoch, 0.8)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs],
                            legend=['content', 'style', 'TV'],
                            ncols=2, figsize=(7, 2.5))
    for epoch in range(num_epochs):
        trainer.zero_grad()
        contents_Y_hat, styles_Y_hat = extract_features(
            X, content_layers, style_layers)
        contents_l, styles_l, tv_l, l = compute_loss(
            X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)
        l.backward()
        trainer.step()
        scheduler.step()
        if (epoch + 1) % 10 == 0:
            animator.axes[1].imshow(postprocess(X))
            animator.add(epoch + 1, [float(sum(contents_l)),
                                     float(sum(styles_l)), float(tv_l)])
    return X
```

```
def train(X, contents_Y, styles_Y, device, lr, num_epochs, lr_decay_epoch):
    X, styles_Y_gram, trainer = get_inits(X, device, lr, styles_Y)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs], ylim=[0, 20],
                            legend=['content', 'style', 'TV'],
                            ncols=2, figsize=(7, 2.5))
    for epoch in range(num_epochs):
        with autograd.record():
            contents_Y_hat, styles_Y_hat = extract_features(
                X, content_layers, style_layers)
            contents_l, styles_l, tv_l, l = compute_loss(
                X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)
        l.backward()
        trainer.step(1)
        if (epoch + 1) % lr_decay_epoch == 0:
            trainer.set_learning_rate(trainer.learning_rate * 0.8)
```

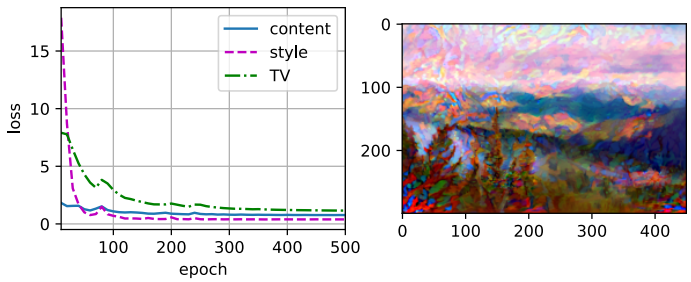
(continues on next page)

(continued from previous page)

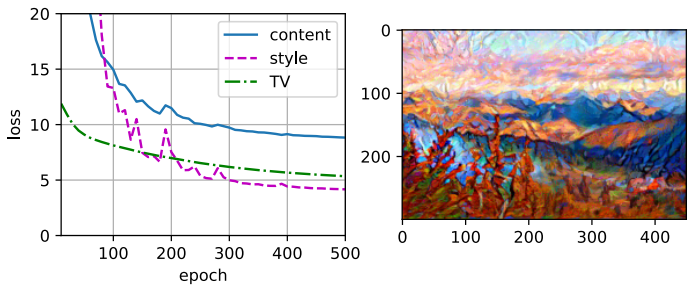
```
        if (epoch + 1) % 10 == 0:
            animator.axes[1].imshow(postprocess(X).asnumpy())
            animator.add(epoch + 1, [float(sum(contents_l)),
                                     float(sum(styles_l)), float(tv_l)])
    return X
```

それでは、モデルの学習を開始する。コンテンツ画像とスタイル画像の高さと幅を 300×450 ピクセルにリスケールする。合成画像の初期化にはコンテンツ画像を用いる。

```
device, image_shape = d2l.try_gpu(), (300, 450)  # PIL Image (h, w)
net = net.to(device)
content_X, contents_Y = get_contents(image_shape, device)
_, styles_Y = get_styles(image_shape, device)
output = train(content_X, contents_Y, styles_Y, device, 0.3, 500, 50)
```




```
device, image_shape = d2l.try_gpu(), (450, 300)
net.collect_params().reset_ctx(device)
content_X, contents_Y = get_contents(image_shape, device)
_, styles_Y = get_styles(image_shape, device)
output = train(content_X, contents_Y, styles_Y, device, 0.9, 500, 50)
```



合成画像がコンテンツ画像の風景や物体を保持しつつ、同時にスタイル画像の色彩を転移していることがわかる。たとえば、合成画像にはスタイル画像に見られるような色の塊がある。それらの一部には、筆致の微妙な質感さえ見られる。

17.12.8 まとめ

- スタイル変換で一般的に用いられる損失関数は 3 つの部分から成る。(i) コンテンツ損失は、合成画像とコンテンツ画像をコンテンツ特徴の面で近づける。(ii) スタイル損失は、合成画像とスタイル画像をスタイル特徴の面で近づける。(iii) 全変動損失は、合成画像中のノイズを減らすのに役立つ。
- 事前学習済み CNN を用いて画像特徴を抽出し、損失関数を最小化することで、学習中に合成画像をモデルパラメータとして継続的に更新できる。
- スタイル層からのスタイル出力を表すためにグラム行列を用いる。

17.12.9 演習

1. 異なるコンテンツ層とスタイル層を選ぶと、出力はどのように変化するか？
2. 損失関数の重みハイパーパラメータを調整しよ。出力はより多くのコンテンツを保持するか、それともノイズが少なくなるか？
3. 異なるコンテンツ画像とスタイル画像を使ってみよ。より興味深い合成画像を作れますか？
4. テキストに対してスタイル変換を適用できるか？ヒント: Hu *et al.* (2022) のサーベイ論文を参照しよ。

17.13 Kaggle における画像分類（CIFAR-10）

これまで、深層学習フレームワークの高レベル API を用いて、画像データセットをテンソル形式で直接取得してきた。しかし、独自の画像データセットはしばしば画像ファイルの形で提供される。この節では、生の画像ファイルから出発し、それらを整理し、読み込み、さらに段階的にテンソル形式へ変換していきる。

私たちは Section 17.1 で CIFAR-10 データセットを扱った。これはコンピュータビジョンにおける重要なデータセットである。この節では、これまでの節で学んだ知識を CIFAR-10 画像分類の Kaggle コンペティションで実践する。コンペティションの Web アドレスは https://www.kaggle.com/c/cifar-10 です

Fig. 17.13.1 はコンペティションの Web ページにある情報を示している。結果を提出するには、Kaggle アカウントを登録する必要がある。

```
import collections
from d2l import torch as d2l
import math
import torch
import torchvision
```

(continues on next page)

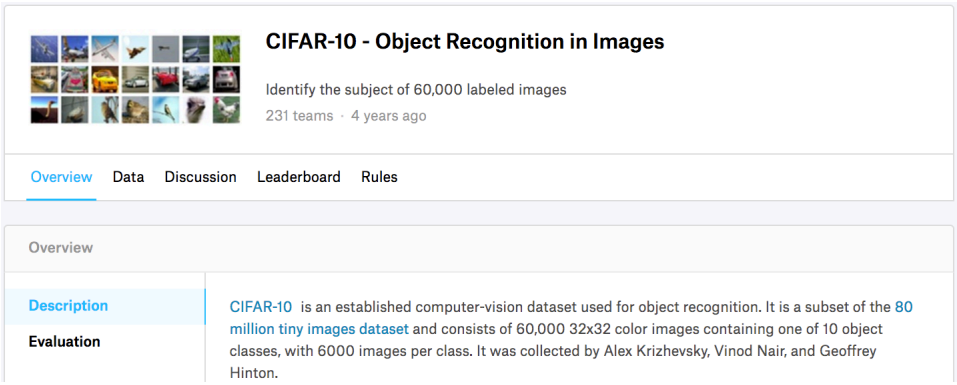


Fig. 17.13.1 CIFAR-10 画像分類コンペティションの Web ページ情報。コンペティションのデータセットは“Data”タブをクリックして取得できる。

(continued from previous page)

```
from torch import nn
import os
import pandas as pd
import shutil
```

```
import collections
from d2l import mxnet as d2l
import math
from mxnet import gluon, init, npx
from mxnet.gluon import nn
import os
import pandas as pd
import shutil

npx.set_np()
```

```
#@save
d2l.DATA_HUB['cifar10_tiny'] = (d2l.DATA_URL + 'kaggle_cifar10_tiny.zip',
                                '2068874e4b9a9f0fb07ebe0ad2b29754449ccacd')

# If you use the full dataset downloaded for the Kaggle competition, set
# `demo` to False
demo = True

if demo:
    data_dir = d2l.download_extract('cifar10_tiny')
else:
    data_dir = '../data/cifar-10/'
```

```
#@save
d2l.DATA_HUB['cifar10_tiny'] = (d2l.DATA_URL + 'kaggle_cifar10_tiny.zip',
                                '2068874e4b9a9f0fb07ebe0ad2b29754449ccacd')
```

(continues on next page)

(continued from previous page)

```
# If you use the full dataset downloaded for the Kaggle competition, set
# `demo` to False
demo = True

if demo:
    data_dir = d2l.download_extract('cifar10_tiny')
else:
    data_dir = '../data/cifar-10/'
```

17.13.1 データセットの取得と整理

コンペティションのデータセットは訓練セットとテストセットに分かれており、それぞれ 50000 枚と 300000 枚の画像を含む。テストセットでは、10000 枚の画像が評価に用いられ、残りの 290000 枚の画像は評価されない。これらは、テストセットの結果を**手作業で**ラベル付けしたものを使って不正に解答することを難しくするために含まれている。このデータセットの画像はすべて png 形式のカラー（RGB チャネル）画像ファイルで、高さと幅はいずれも 32 ピクセルである。画像は全部で 10 クラス、すなわち airplane, car, bird, cat, deer, dog, frog, horse, boat, truck を含む。Fig. 17.13.1 の左上には、データセット中の airplane, car, bird の画像がいくつか示されている。

データセットのダウンロード

Kaggle にログインした後、Fig. 17.13.1 に示した CIFAR-10 画像分類コンペティションの Web ページで“Data”タブをクリックし、“Download All”ボタンを押してデータセットをダウンロードできる。ダウンロードしたファイルを `../data` に解凍し、その中の `train.7z` と `test.7z` をさらに解凍すると、データセット全体は次のパスにある。

- `../data/cifar-10/train/[1-50000].png`
- `../data/cifar-10/test/[1-300000].png`
- `../data/cifar-10/trainLabels.csv`
- `../data/cifar-10/sampleSubmission.csv`

ここで `train` と `test` ディレクトリにはそれぞれ訓練画像とテスト画像が含まれ、`trainLabels.csv` は訓練画像のラベルを提供し、`sample_submission.csv` は提出用のサンプルファイルである。

始めやすくするために、データセットの小規模サンプルを提供する。これは最初の 1000 枚の訓練画像と、ランダムに選んだ 5 枚のテスト画像を含む。Kaggle コンペティションの完全なデータセットを使うには、次の `demo` 変数を `False` に設定する必要がある。

```
#@save
d2l.DATA_HUB['cifar10_tiny'] = (d2l.DATA_URL + 'kaggle_cifar10_tiny.zip',
                                '2068874e4b9a9f0fb07ebe0ad2b29754449ccacd')

# If you use the full dataset downloaded for the Kaggle competition, set
# `demo` to False
demo = True

if demo:
    data_dir = d2l.download_extract('cifar10_tiny')
else:
    data_dir = '../data/cifar-10/'
```

```
Downloading ../data/kaggle_cifar10_tiny.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/kaggle_cifar10_tiny.zip...
```


```
#@save
d2l.DATA_HUB['cifar10_tiny'] = (d2l.DATA_URL + 'kaggle_cifar10_tiny.zip',
                                '2068874e4b9a9f0fb07ebe0ad2b29754449ccacd')

# If you use the full dataset downloaded for the Kaggle competition, set
# `demo` to False
demo = True

if demo:
    data_dir = d2l.download_extract('cifar10_tiny')
else:
    data_dir = '../data/cifar-10/'
```

```
Downloading ../data/kaggle_cifar10_tiny.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/kaggle_cifar10_tiny.zip...
```


```
#@save
def read_csv_labels(fname):
    """Read `fname` to return a filename to label dictionary."""
    with open(fname, 'r') as f:
        # Skip the file header line (column name)
        lines = f.readlines()[1:]
    tokens = [l.rstrip().split(',') for l in lines]
    return dict(((name, label) for name, label in tokens))

labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
print('# training examples:', len(labels))
print('# classes:', len(set(labels.values())))
```

```
#@save
def read_csv_labels(fname):
    """Read `fname` to return a filename to label dictionary."""
    with open(fname, 'r') as f:
        # Skip the file header line (column name)
```

(continues on next page)

(continued from previous page)

```
        lines = f.readlines()[1:]
    tokens = [l.rstrip().split(',') for l in lines]
    return dict(((name, label) for name, label in tokens))

labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
print('# training examples:', len(labels))
print('# classes:', len(set(labels.values())))
```

データセットの整理

モデルの訓練とテストを容易にするために、データセットを整理する必要がある。まず csv ファイルからラベルを読み込みよう。次の関数は、拡張子を除いたファイル名をそのラベルに対応付ける辞書を返す。

```
#@save
def read_csv_labels(fname):
    """Read `fname` to return a filename to label dictionary."""
    with open(fname, 'r') as f:
        # Skip the file header line (column name)
        lines = f.readlines()[1:]
    tokens = [l.rstrip().split(',') for l in lines]
    return dict(((name, label) for name, label in tokens))

labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
print('# training examples:', len(labels))
print('# classes:', len(set(labels.values())))
```

```
# training examples: 1000
# classes: 10
```

```
#@save
def read_csv_labels(fname):
    """Read `fname` to return a filename to label dictionary."""
    with open(fname, 'r') as f:
        # Skip the file header line (column name)
        lines = f.readlines()[1:]
    tokens = [l.rstrip().split(',') for l in lines]
    return dict(((name, label) for name, label in tokens))

labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
print('# training examples:', len(labels))
print('# classes:', len(set(labels.values())))
```

```
# training examples: 1000
# classes: 10
```

```
#@save
```

(continues on next page)

(continued from previous page)

```
def copyfile(filename, target_dir):
    """Copy a file into a target directory."""
    os.makedirs(target_dir, exist_ok=True)
    shutil.copy(filename, target_dir)

#@save
def reorg_train_valid(data_dir, labels, valid_ratio):
    """Split the validation set out of the original training set."""
    # The number of examples of the class that has the fewest examples in the
    # training dataset
    n = collections.Counter(labels.values()).most_common()[-1][1]
    # The number of examples per class for the validation set
    n_valid_per_label = max(1, math.floor(n * valid_ratio))
    label_count = {}
    for train_file in os.listdir(os.path.join(data_dir, 'train')):
        label = labels[train_file.split('.')[0]]
        fname = os.path.join(data_dir, 'train', train_file)
        copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                     'train_valid', label))
        if label not in label_count or label_count[label] < n_valid_per_label:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'train', label))
    return n_valid_per_label
```

```
#@save
def copyfile(filename, target_dir):
    """Copy a file into a target directory."""
    os.makedirs(target_dir, exist_ok=True)
    shutil.copy(filename, target_dir)

#@save
def reorg_train_valid(data_dir, labels, valid_ratio):
    """Split the validation set out of the original training set."""
    # The number of examples of the class that has the fewest examples in the
    # training dataset
    n = collections.Counter(labels.values()).most_common()[-1][1]
    # The number of examples per class for the validation set
    n_valid_per_label = max(1, math.floor(n * valid_ratio))
    label_count = {}
    for train_file in os.listdir(os.path.join(data_dir, 'train')):
        label = labels[train_file.split('.')[0]]
        fname = os.path.join(data_dir, 'train', train_file)
        copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                     'train_valid', label))
        if label not in label_count or label_count[label] < n_valid_per_label:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
```

(continues on next page)

(continued from previous page)

```
                                        'train', label))
    return n_valid_per_label
```

次に、reorg_train_valid 関数を定義して、元の訓練セットから検証セットを分割して取り出す。この関数の引数 valid_ratio は、検証セットの例数と元の訓練セットの例数の比率である。より具体的には、最も例数の少ないクラスの画像数を n、比率を r とすると、検証セットは各クラスから $\max(\lfloor nr \rfloor, 1)$ 枚の画像を分割して取り出す。例として valid_ratio=0.1 を使いよう。元の訓練セットには 50000 枚の画像があるので、train_valid_test/train には 45000 枚の画像が訓練用として使われ、残りの 5000 枚は train_valid_test/valid に検証セットとして分割される。データセットを整理した後は、同じクラスの画像が同じフォルダの下に配置される。

```
#@save
def copyfile(filename, target_dir):
    """Copy a file into a target directory."""
    os.makedirs(target_dir, exist_ok=True)
    shutil.copy(filename, target_dir)

#@save
def reorg_train_valid(data_dir, labels, valid_ratio):
    """Split the validation set out of the original training set."""
    # The number of examples of the class that has the fewest examples in the
    # training dataset
    n = collections.Counter(labels.values()).most_common()[-1][1]
    # The number of examples per class for the validation set
    n_valid_per_label = max(1, math.floor(n * valid_ratio))
    label_count = {}
    for train_file in os.listdir(os.path.join(data_dir, 'train')):
        label = labels[train_file.split('.')[0]]
        fname = os.path.join(data_dir, 'train', train_file)
        copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                     'train_valid', label))
        if label not in label_count or label_count[label] < n_valid_per_label:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'train', label))
    return n_valid_per_label
```

```
#@save
def copyfile(filename, target_dir):
    """Copy a file into a target directory."""
    os.makedirs(target_dir, exist_ok=True)
    shutil.copy(filename, target_dir)

#@save
def reorg_train_valid(data_dir, labels, valid_ratio):
```

(continues on next page)

(continued from previous page)

```
    """Split the validation set out of the original training set."""
    # The number of examples of the class that has the fewest examples in the
    # training dataset
    n = collections.Counter(labels.values()).most_common()[-1][1]
    # The number of examples per class for the validation set
    n_valid_per_label = max(1, math.floor(n * valid_ratio))
    label_count = {}
    for train_file in os.listdir(os.path.join(data_dir, 'train')):
        label = labels[train_file.split('.')[0]]
        fname = os.path.join(data_dir, 'train', train_file)
        copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                     'train_valid', label))
        if label not in label_count or label_count[label] < n_valid_per_label:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'train', label))
    return n_valid_per_label
```

```
#@save
def reorg_test(data_dir):
    """Organize the testing set for data loading during prediction."""
    for test_file in os.listdir(os.path.join(data_dir, 'test')):
        copyfile(os.path.join(data_dir, 'test', test_file),
                 os.path.join(data_dir, 'train_valid_test', 'test',
                              'unknown'))
```

```
#@save
def reorg_test(data_dir):
    """Organize the testing set for data loading during prediction."""
    for test_file in os.listdir(os.path.join(data_dir, 'test')):
        copyfile(os.path.join(data_dir, 'test', test_file),
                 os.path.join(data_dir, 'train_valid_test', 'test',
                              'unknown'))
```

以下の reorg_test 関数は、予測時のデータ読み込みのためにテストセットを整理する。

```
#@save
def reorg_test(data_dir):
    """Organize the testing set for data loading during prediction."""
    for test_file in os.listdir(os.path.join(data_dir, 'test')):
        copyfile(os.path.join(data_dir, 'test', test_file),
                 os.path.join(data_dir, 'train_valid_test', 'test',
                              'unknown'))
```

```
#@save
def reorg_test(data_dir):
```

(continues on next page)

(continued from previous page)

```
    """Organize the testing set for data loading during prediction."""
    for test_file in os.listdir(os.path.join(data_dir, 'test')):
        copyfile(os.path.join(data_dir, 'test', test_file),
                 os.path.join(data_dir, 'train_valid_test', 'test',
                              'unknown'))
```

```
def reorg_cifar10_data(data_dir, valid_ratio):
    labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
    reorg_train_valid(data_dir, labels, valid_ratio)
    reorg_test(data_dir)
```


```
def reorg_cifar10_data(data_dir, valid_ratio):
    labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
    reorg_train_valid(data_dir, labels, valid_ratio)
    reorg_test(data_dir)
```


最後に、上で定義した read_csv_labels、reorg_train_valid、reorg_test 関数を呼び出すための関数を使う。


```
def reorg_cifar10_data(data_dir, valid_ratio):
    labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
    reorg_train_valid(data_dir, labels, valid_ratio)
    reorg_test(data_dir)
```



```
def reorg_cifar10_data(data_dir, valid_ratio):
    labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
    reorg_train_valid(data_dir, labels, valid_ratio)
    reorg_test(data_dir)
```


```
batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_cifar10_data(data_dir, valid_ratio)
```


```
batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_cifar10_data(data_dir, valid_ratio)
```


ここでは、データセットの小規模サンプルに対してバッチサイズを 32 に設定している。Kaggle コンペティションの完全なデータセットで訓練・テストする場合は、batch_size は 128 のようなより大きい整数に設定すべきである。ハイパーパラメータ調整のために、訓練例の 10% を検証セットとして分割する。


```
batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_cifar10_data(data_dir, valid_ratio)
```

17.13.2 画像拡張

過学習に対処するために画像拡張を用いる。たとえば、訓練中に画像をランダムに左右反転できる。また、カラー画像の 3 つの RGB チャネルに対して標準化を行うこともできる。以下に、調整可能な操作のいくつかを示す。

```
transform_train = torchvision.transforms.Compose([
    # Scale the image up to a square of 40 pixels in both height and width
    torchvision.transforms.Resize(40),
    # Randomly crop a square image of 40 pixels in both height and width to
    # produce a small square of 0.64 to 1 times the area of the original
    # image, and then scale it to a square of 32 pixels in both height and
    # width
    torchvision.transforms.RandomResizedCrop(32, scale=(0.64, 1.0),
                                             ratio=(1.0, 1.0)),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    # Standardize each channel of the image
    torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                     [0.2023, 0.1994, 0.2010])])
```

```
transform_train = gluon.data.vision.transforms.Compose([
    # Scale the image up to a square of 40 pixels in both height and width
    gluon.data.vision.transforms.Resize(40),
    # Randomly crop a square image of 40 pixels in both height and width to
    # produce a small square of 0.64 to 1 times the area of the original
    # image, and then scale it to a square of 32 pixels in both height and
    # width
    gluon.data.vision.transforms.RandomResizedCrop(32, scale=(0.64, 1.0),
                                                   ratio=(1.0, 1.0)),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    gluon.data.vision.transforms.ToTensor(),
    # Standardize each channel of the image
    gluon.data.vision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                           [0.2023, 0.1994, 0.2010])])
```

テスト時には、評価結果にランダム性が入らないように、画像の標準化のみを行う。

```
transform_test = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                     [0.2023, 0.1994, 0.2010])])
```

```
transform_test = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.ToTensor(),
    gluon.data.vision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                           [0.2023, 0.1994, 0.2010])])
```

17.13.3 データセットの読み込み

次に、整理済みの生画像ファイルからなるデータセットを読み込みます。各例は画像とラベルを含む。

```
train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_train) for folder in ['train', 'train_valid']]

valid_ds, test_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_test) for folder in ['valid', 'test']]
```

```
train_ds, valid_ds, train_valid_ds, test_ds = [
    gluon.data.vision.ImageFolderDataset(
        os.path.join(data_dir, 'train_valid_test', folder))
    for folder in ['train', 'valid', 'train_valid', 'test']]
```

訓練中には、上で定義したすべての画像拡張操作を指定する必要がある。検証セットをハイパーパラメータ調整中のモデル評価に用いるときは、画像拡張によるランダム性を入れてはいけない。最終予測の前には、すべてのラベル付きデータを最大限に活用するために、訓練セットと検証セットを結合したデータでモデルを再訓練する。

```
train_iter, train_valid_iter = [torch.utils.data.DataLoader(
    dataset, batch_size, shuffle=True, drop_last=True)
    for dataset in (train_ds, train_valid_ds)]

valid_iter = torch.utils.data.DataLoader(valid_ds, batch_size, shuffle=False,
                                         drop_last=True)

test_iter = torch.utils.data.DataLoader(test_ds, batch_size, shuffle=False,
                                        drop_last=False)
```

```
train_iter, train_valid_iter = [gluon.data.DataLoader(
    dataset.transform_first(transform_train), batch_size, shuffle=True,
    last_batch='discard') for dataset in (train_ds, train_valid_ds)]

valid_iter = gluon.data.DataLoader(
    valid_ds.transform_first(transform_test), batch_size, shuffle=False,
    last_batch='discard')

test_iter = gluon.data.DataLoader(
    test_ds.transform_first(transform_test), batch_size, shuffle=False,
    last_batch='keep')
```

17.13.4 モデルの定義

Section 11.6 で説明した ResNet-18 モデルを定義する。

```
def get_net():
    num_classes = 10
    net = d2l.resnet18(num_classes, 3)
    return net

loss = nn.CrossEntropyLoss(reduction="none")
```

```
class Residual(nn.HybridBlock):
    def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                               strides=strides)
        self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                   strides=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def hybrid_forward(self, F, X):
        Y = F.npx.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return F.npx.relu(Y + X)
```

17.13.5 訓練関数の定義

モデルの検証セット上での性能に基づいて、モデルを選択し、ハイパーパラメータを調整する。以下では、モデル訓練関数 `train` を定義する。

```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    trainer = torch.optim.SGD(net.parameters(), lr=lr, momentum=0.9,
                              weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_period, lr_decay)
    num_batches, timer = len(train_iter), d2l.Timer()
    legend = ['train loss', 'train acc']
    if valid_iter is not None:
        legend.append('valid acc')
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=legend)
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    for epoch in range(num_epochs):
        net.train()
        metric = d2l.Accumulator(3)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = d2l.train_batch_ch13(net, features, labels,
                                          loss, trainer, devices)
```

(continues on next page)

(continued from previous page)

```
            metric.add(l, acc, labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[2],
                              None))
        if valid_iter is not None:
            valid_acc = d2l.evaluate_accuracy_gpu(net, valid_iter)
            animator.add(epoch + 1, (None, None, valid_acc))
        scheduler.step()
    measures = (f'train loss {metric[0] / metric[2]:.3f}, '
                f'train acc {metric[1] / metric[2]:.3f}')
    if valid_iter is not None:
        measures += f', valid acc {valid_acc:.3f}'
    print(measures + f'\n{metric[2] * num_epochs / timer.sum():.1f}'
          f' examples/sec on {str(devices)}')
```

```
def resnet18(num_classes):
    net = nn.HybridSequential()
    net.add(nn.Conv2D(64, kernel_size=3, strides=1, padding=1),
            nn.BatchNorm(), nn.Activation('relu'))

    def resnet_block(num_channels, num_residuals, first_block=False):
        blk = nn.HybridSequential()
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
            else:
                blk.add(Residual(num_channels))
        return blk

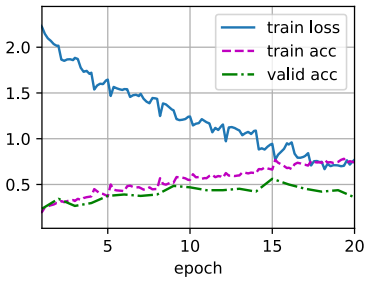
    net.add(resnet_block(64, 2, first_block=True),
            resnet_block(128, 2),
            resnet_block(256, 2),
            resnet_block(512, 2))
    net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
    return net
```

17.13.6 モデルの訓練と検証

これで、モデルを訓練し検証できる。以下のハイパーパラメータはすべて調整可能である。たとえば、エポック数を増やすことができる。lr_period と lr_decay をそれぞれ 4 と 0.9 に設定すると、最適化アルゴリズムの学習率は 4 エポックごとに 0.9 倍される。説明の便宜上、ここでは 20 エポックだけ訓練する。

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 20, 2e-4, 5e-4
lr_period, lr_decay, net = 4, 0.9, get_net()
net(next(iter(train_iter))[0])
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

```
train loss 0.771, train acc 0.740, valid acc 0.359
986.7 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```




```
def get_net(devices):
    num_classes = 10
    net = resnet18(num_classes)
    net.initialize(ctx=devices, init=init.Xavier())
    return net

loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

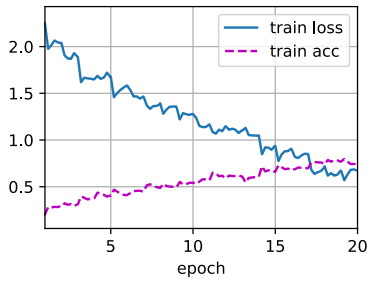
17.13.7 テストセットの分類と Kaggle への結果提出

ハイパーパラメータを調整して有望なモデルを得たら、すべてのラベル付きデータ（検証セットを含む）を使ってモデルを再訓練し、テストセットを分類する。

```
net, preds = get_net(), []
net(next(iter(train_valid_iter))[0])
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

for X, _ in test_iter:
    y_hat = net(X.to(devices[0]))
    preds.extend(y_hat.argmax(dim=1).type(torch.int32).cpu().numpy())
sorted_ids = list(range(1, len(test_ds) + 1))
sorted_ids.sort(key=lambda x: str(x))
df = pd.DataFrame({'id': sorted_ids, 'label': preds})
df['label'] = df['label'].apply(lambda x: train_valid_ds.classes[x])
df.to_csv('submission.csv', index=False)
```

```
train loss 0.676, train acc 0.746
1136.6 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```




```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})
    num_batches, timer = len(train_iter), d2l.Timer()
    legend = ['train loss', 'train acc']
    if valid_iter is not None:
        legend.append('valid acc')
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=legend)
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)
        if epoch > 0 and epoch % lr_period == 0:
            trainer.set_learning_rate(trainer.learning_rate * lr_decay)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = d2l.train_batch_ch13(
                net, features, labels.astype('float32'), loss, trainer,
                devices, d2l.split_batch)
            metric.add(l, acc, labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[2],
                              None))
        if valid_iter is not None:
            valid_acc = d2l.evaluate_accuracy_gpus(net, valid_iter,
                                                   d2l.split_batch)
            animator.add(epoch + 1, (None, None, valid_acc))
    measures = (f'train loss {metric[0] / metric[2]:.3f}, '
                f'train acc {metric[1] / metric[2]:.3f}')
    if valid_iter is not None:
        measures += f', valid acc {valid_acc:.3f}'
    print(measures + f'\n{metric[2] * num_epochs / timer.sum():.1f}'
          f' examples/sec on {str(devices)}')
```

上のコードは submission.csv ファイルを生成する。その形式は Kaggle コンペティションの要件を満たしている。Kaggle への結果提出方法は Section 8.7 の場合と似ている。

17.13.8 まとめ

- 生の画像ファイルを必要な形式に整理すれば、それらを含むデータセットを読み込める。
- 画像分類コンペティションでは、畳み込みニューラルネットワークと画像拡張を利用できる。

17.13.9 演習

1. この Kaggle コンペティションで CIFAR-10 の完全なデータセットを使いなさい。ハイパーパラメータを `batch_size = 128`, `num_epochs = 100`, `lr = 0.1`, `lr_period = 50`, `lr_decay = 0.1` に設定しなさい。このコンペティションでどの程度の精度と順位を達成できるか調べなさい。さらに改善できるか？
2. 画像拡張を使わない場合、どの程度の精度が得られるか？

17.14 Kaggle における犬種識別（ImageNet Dogs）

この節では、Kaggle 上の犬種識別問題を実践する。このコンペティションの Web アドレスは https://www.kaggle.com/c/dog-breed-identification です

このコンペティションでは、120 種類の犬種を識別する。実際、このコンペティションのデータセットは ImageNet データセットのサブセットである。Section 17.13 における CIFAR-10 データセットの画像とは異なり、ImageNet データセットの画像は縦横ともにより大きく、サイズもさまざまである。Fig. 17.14.1 はコンペティションの Web ページの情報を示している。結果を提出するには Kaggle アカウントが必要である。

```
from d2l import torch as d2l
import torch
import torchvision
from torch import nn
import os
```

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, npx
from mxnet.gluon import nn
import os

npx.set_np()
```

```
#@save
d2l.DATA_HUB['dog_tiny'] = (d2l.DATA_URL + 'kaggle_dog_tiny.zip',
                            '0cb91d09b814ecdc07b50f31f8dcad3e81d6a86d')
```

(continues on next page)

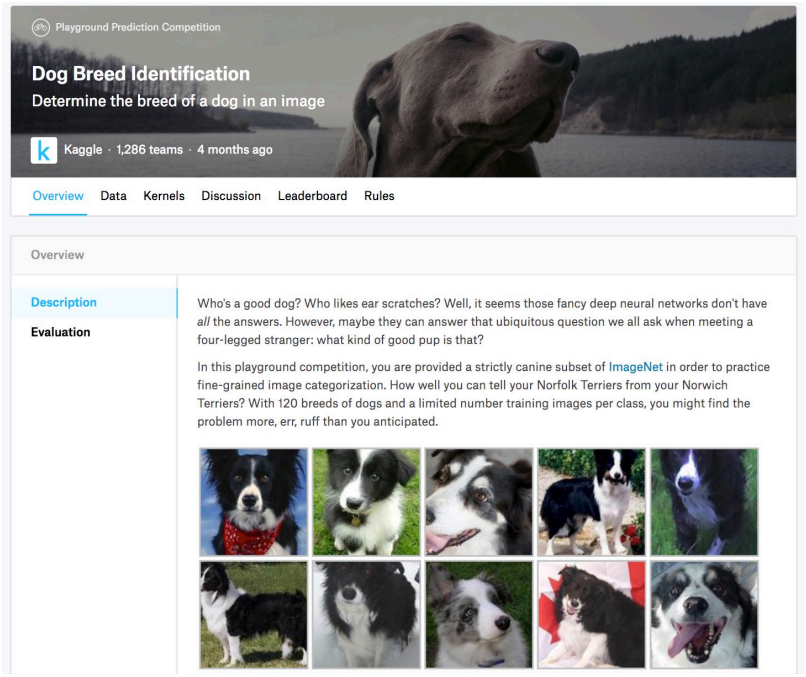


Fig. 17.14.1 犬種識別コンペティションの Web サイト。コンペティションのデータセットは“Data”タブをクリックして取得できる。

(continued from previous page)

```
# If you use the full dataset downloaded for the Kaggle competition, change
# the variable below to `False`
demo = True
if demo:
    data_dir = d2l.download_extract('dog_tiny')
else:
    data_dir = os.path.join('..', 'data', 'dog-breed-identification')
```

```
#@save
d2l.DATA_HUB['dog_tiny'] = (d2l.DATA_URL + 'kaggle_dog_tiny.zip',
                            '0cb91d09b814ecdc07b50f31f8dcad3e81d6a86d')

# If you use the full dataset downloaded for the Kaggle competition, change
# the variable below to `False`
demo = True
if demo:
    data_dir = d2l.download_extract('dog_tiny')
else:
    data_dir = os.path.join('..', 'data', 'dog-breed-identification')
```

17.14.1 データセットの取得と整理

コンペティションのデータセットは訓練セットとテストセットに分かれており、それぞれ 10222 枚と 10357 枚の JPEG 画像からなる。各画像は 3 つの RGB（カラー）チャネルを持つ。訓練データセットには、ラブラドール、プードル、ダックスフント、サモエド、ハスキー、チワワ、ヨークシャー・テリアなど、120 種類の犬種が含まれている。

データセットのダウンロード

Kaggle にログインした後、Fig. 17.14.1 に示したコンペティションの Web ページで“Data”タブをクリックし、“Download All”ボタンを押してデータセットをダウンロードできる。ダウンロードしたファイルを `../data` に解凍すると、データセット全体は次のパスにある。

- ../data/dog-breed-identification/labels.csv
- ../data/dog-breed-identification/sample_submission.csv
- ../data/dog-breed-identification/train
- ../data/dog-breed-identification/test

上の構成は Section 17.13 の CIFAR-10 コンペティションの構成と似ていることに気づいたかもしれない。そこでは `train/` と `test/` フォルダにそれぞれ訓練用とテスト用の犬画像が含まれ、`labels.csv` には訓練画像のラベルが含まれている。同様に、始めやすいように、上で述べたデータセットの小さなサンプルを提供している: `train_valid_test_tiny.zip`。Kaggle コンペティション用に完全なデータセットを使う場合は、下の `demo` 変数を `False` に変更する必要がある。

```
#@save
d2l.DATA_HUB['dog_tiny'] = (d2l.DATA_URL + 'kaggle_dog_tiny.zip',
                            '0cb91d09b814ecdc07b50f31f8dcad3e81d6a86d')

# If you use the full dataset downloaded for the Kaggle competition, change
# the variable below to `False`
demo = True
if demo:
    data_dir = d2l.download_extract('dog_tiny')
else:
    data_dir = os.path.join('..', 'data', 'dog-breed-identification')
```

```
Downloading ../data/kaggle_dog_tiny.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/kaggle_dog_tiny.zip...
```

```
#@save
d2l.DATA_HUB['dog_tiny'] = (d2l.DATA_URL + 'kaggle_dog_tiny.zip',
                            '0cb91d09b814ecdc07b50f31f8dcad3e81d6a86d')
```

(continues on next page)

(continued from previous page)

```
# If you use the full dataset downloaded for the Kaggle competition, change
# the variable below to `False`
demo = True
if demo:
    data_dir = d2l.download_extract('dog_tiny')
else:
    data_dir = os.path.join('..', 'data', 'dog-breed-identification')
```

```
Downloading ../data/kaggle_dog_tiny.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/kaggle_dog_tiny.zip...
```

```
def reorg_dog_data(data_dir, valid_ratio):
    labels = d2l.read_csv_labels(os.path.join(data_dir, 'labels.csv'))
    d2l.reorg_train_valid(data_dir, labels, valid_ratio)
    d2l.reorg_test(data_dir)


batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_dog_data(data_dir, valid_ratio)
```

```
def reorg_dog_data(data_dir, valid_ratio):
    labels = d2l.read_csv_labels(os.path.join(data_dir, 'labels.csv'))
    d2l.reorg_train_valid(data_dir, labels, valid_ratio)
    d2l.reorg_test(data_dir)


batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_dog_data(data_dir, valid_ratio)
```

データセットの整理

Section 17.13 で行ったのと同様に、データセットを整理できる。つまり、元の訓練セットから検証セットを分割し、画像をラベルごとにまとめたサブフォルダへ移動する。

以下の reorg_dog_data 関数は訓練データのラベルを読み込み、検証セットを分割し、訓練セットを整理する。

```
def reorg_dog_data(data_dir, valid_ratio):
    labels = d2l.read_csv_labels(os.path.join(data_dir, 'labels.csv'))
    d2l.reorg_train_valid(data_dir, labels, valid_ratio)
    d2l.reorg_test(data_dir)


batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_dog_data(data_dir, valid_ratio)
```

17.14.2 画像拡張

この犬種データセットは ImageNet データセットのサブセットであり、その画像は Section 17.13 の CIFAR-10 データセットの画像よりも大きいことを思い出しよ。以下では、比較的大きな画像に対して有用と思われるいくつかの画像拡張操作を示す。

```
transform_train = torchvision.transforms.Compose([
    # Randomly crop the image to obtain an image with an area of 0.08 to 1 of
    # the original area and height-to-width ratio between 3/4 and 4/3. Then,
    # scale the image to create a new 224 x 224 image
    torchvision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                             ratio=(3.0/4.0, 4.0/3.0)),
    torchvision.transforms.RandomHorizontalFlip(),
    # Randomly change the brightness, contrast, and saturation
    torchvision.transforms.ColorJitter(brightness=0.4,
                                       contrast=0.4,
                                       saturation=0.4),
    # Add random noise
    torchvision.transforms.ToTensor(),
    # Standardize each channel of the image
    torchvision.transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])])
```

```
transform_train = gluon.data.vision.transforms.Compose([
    # Randomly crop the image to obtain an image with an area of 0.08 to 1 of
    # the original area and height-to-width ratio between 3/4 and 4/3. Then,
    # scale the image to create a new 224 x 224 image
    gluon.data.vision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                                   ratio=(3.0/4.0, 4.0/3.0)),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    # Randomly change the brightness, contrast, and saturation
    gluon.data.vision.transforms.RandomColorJitter(brightness=0.4,
                                                   contrast=0.4,
                                                   saturation=0.4),
    # Add random noise
    gluon.data.vision.transforms.RandomLighting(0.1),
    gluon.data.vision.transforms.ToTensor(),
    # Standardize each channel of the image
    gluon.data.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                           [0.229, 0.224, 0.225])])
```

予測時には、ランダム性のない画像前処理操作のみを使用する。

```
transform_test = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    # Crop a 224 x 224 square area from the center of the image
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])])
```

```
transform_test = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.Resize(256),
    # Crop a 224 x 224 square area from the center of the image
    gluon.data.vision.transforms.CenterCrop(224),
    gluon.data.vision.transforms.ToTensor(),
    gluon.data.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                           [0.229, 0.224, 0.225])])
```

17.14.3 データセットの読み込み

Section 17.13 と同様に、生の画像ファイルからなる整理済みデータセットを読み込める。

```
train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_train) for folder in ['train', 'train_valid']]

valid_ds, test_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_test) for folder in ['valid', 'test']]
```

```
train_ds, valid_ds, train_valid_ds, test_ds = [
    gluon.data.vision.ImageFolderDataset(
        os.path.join(data_dir, 'train_valid_test', folder))
    for folder in ('train', 'valid', 'train_valid', 'test')]
```

以下では、データ反復子のインスタンスを Section 17.13 と同じ方法で作成する。

```
train_iter, train_valid_iter = [torch.utils.data.DataLoader(
    dataset, batch_size, shuffle=True, drop_last=True)
    for dataset in (train_ds, train_valid_ds)]

valid_iter = torch.utils.data.DataLoader(valid_ds, batch_size, shuffle=False,
                                         drop_last=True)

test_iter = torch.utils.data.DataLoader(test_ds, batch_size, shuffle=False,
                                        drop_last=False)
```

```
train_iter, train_valid_iter = [gluon.data.DataLoader(
    dataset.transform_first(transform_train), batch_size, shuffle=True,
    last_batch='discard') for dataset in (train_ds, train_valid_ds)]

valid_iter = gluon.data.DataLoader(
    valid_ds.transform_first(transform_test), batch_size, shuffle=False,
    last_batch='discard')

test_iter = gluon.data.DataLoader(
    test_ds.transform_first(transform_test), batch_size, shuffle=False,
    last_batch='keep')
```

17.14.4 事前学習済みモデルのファインチューニング

繰り返しになるが、このコンペティションのデータセットは ImageNet データセットのサブセットである。したがって、Section 17.2 で議論した手法を用いて、完全な ImageNet データセットで事前学習されたモデルを選び、そこから画像特徴を抽出して、それをカスタムの小規模出力ネットワークに入力できる。深層学習フレームワークの高レベル API には、ImageNet データセットで事前学習された多様なモデルが用意されている。ここでは、事前学習済みの ResNet-34 モデルを選び、このモデルの出力層の入力（すなわち抽出された特徴量）をそのまま再利用する。その後、元の出力層を、たとえば 2 層の全結合層を重ねたような、学習可能な小さなカスタム出力ネットワークに置き換える。Section 17.2 の実験とは異なり、以下では特徴抽出に用いる事前学習済みモデルを再学習しない。これにより、学習時間と勾配を保存するためのメモリが削減される。

完全な ImageNet データセットの 3 つの RGB チャネルの平均と標準偏差を用いて画像を標準化したことを思い出しよ。実際、これは ImageNet 上の事前学習済みモデルによる標準化操作とも一致している。

```
def get_net(devices):
    finetune_net = nn.Sequential()
    finetune_net.features = torchvision.models.resnet34(pretrained=True)
    # Define a new output network (there are 120 output categories)
    finetune_net.output_new = nn.Sequential(nn.Linear(1000, 256),
                                            nn.ReLU(),
                                            nn.Linear(256, 120))
    # Move the model to devices
    finetune_net = finetune_net.to(devices[0])
    # Freeze parameters of feature layers
    for param in finetune_net.features.parameters():
        param.requires_grad = False
    return finetune_net
```

```
def get_net(devices):
    finetune_net = gluon.model_zoo.vision.resnet34_v2(pretrained=True)
    # Define a new output network
    finetune_net.output_new = nn.HybridSequential(prefix='')
    finetune_net.output_new.add(nn.Dense(256, activation='relu'))
    # There are 120 output categories
    finetune_net.output_new.add(nn.Dense(120))
    # Initialize the output network
    finetune_net.output_new.initialize(init.Xavier(), ctx=devices)
    # Distribute the model parameters to the CPUs or GPUs used for computation
    finetune_net.collect_params().reset_ctx(devices)
    return finetune_net
```

損失を計算する前に、まず事前学習済みモデルの出力層の入力、すなわち抽出された特徴量を取得する。その後、この特徴量を小さなカスタム出力ネットワークへの入力として用い、損失を計算する。

```
loss = nn.CrossEntropyLoss(reduction='none')

def evaluate_loss(data_iter, net, devices):
    l_sum, n = 0.0, 0
    for features, labels in data_iter:
        features, labels = features.to(devices[0]), labels.to(devices[0])
        outputs = net(features)
        l = loss(outputs, labels)
        l_sum += l.sum()
        n += labels.numel()
    return l_sum / n
```

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()

def evaluate_loss(data_iter, net, devices):
    l_sum, n = 0.0, 0
    for features, labels in data_iter:
        X_shards, y_shards = d2l.split_batch(features, labels, devices)
        output_features = [net.features(X_shard) for X_shard in X_shards]
        outputs = [net.output_new(feature) for feature in output_features]
        ls = [loss(output, y_shard).sum() for output, y_shard
              in zip(outputs, y_shards)]
        l_sum += sum([float(l.sum()) for l in ls])
        n += labels.size
    return l_sum / n
```

17.14.5 学習関数の定義

検証セット上でのモデルの性能に基づいて、モデルを選択し、ハイパーパラメータを調整する。モデル学習関数 `train` は小さなカスタム出力ネットワークのパラメータのみを反復する。

```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    # Only train the small custom output network
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    trainer = torch.optim.SGD((param for param in net.parameters()
                               if param.requires_grad), lr=lr,
                              momentum=0.9, weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_period, lr_decay)
    num_batches, timer = len(train_iter), d2l.Timer()
    legend = ['train loss']
    if valid_iter is not None:
        legend.append('valid loss')
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=legend)
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(2)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            features, labels = features.to(devices[0]), labels.to(devices[0])
            trainer.zero_grad()
```

(continues on next page)

(continued from previous page)

```
            output = net(features)
            l = loss(output, labels).sum()
            l.backward()
            trainer.step()
            metric.add(l, labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[1], None))
        measures = f'train loss {metric[0] / metric[1]:.3f}'
        if valid_iter is not None:
            valid_loss = evaluate_loss(valid_iter, net, devices)
            animator.add(epoch + 1, (None, valid_loss.detach().cpu()))
        scheduler.step()
    if valid_iter is not None:
        measures += f', valid loss {valid_loss:.3f}'
    print(measures + f'\n{metric[1] * num_epochs / timer.sum():.1f}'
          f' examples/sec on {str(devices)}')
```

```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    # Only train the small custom output network
    trainer = gluon.Trainer(net.output_new.collect_params(), 'sgd',
                            {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})
    num_batches, timer = len(train_iter), d2l.Timer()
    legend = ['train loss']
    if valid_iter is not None:
        legend.append('valid loss')
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=legend)
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(2)
        if epoch > 0 and epoch % lr_period == 0:
            trainer.set_learning_rate(trainer.learning_rate * lr_decay)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            X_shards, y_shards = d2l.split_batch(features, labels, devices)
            output_features = [net.features(X_shard) for X_shard in X_shards]
            with autograd.record():
                outputs = [net.output_new(feature)
                           for feature in output_features]
                ls = [loss(output, y_shard).sum() for output, y_shard
                      in zip(outputs, y_shards)]
            for l in ls:
                l.backward()
            trainer.step(batch_size)
            metric.add(sum([float(l.sum()) for l in ls]), labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[1], None))
        if valid_iter is not None:
            valid_loss = evaluate_loss(valid_iter, net, devices)
            animator.add(epoch + 1, (None, valid_loss))
```

(continues on next page)

(continued from previous page)

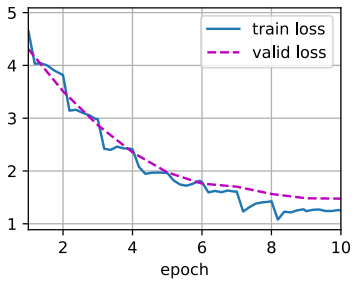
```
    measures = f'train loss {metric[0] / metric[1]:.3f}'
    if valid_iter is not None:
        measures += f', valid loss {valid_loss:.3f}'
    print(measures + f'\n{metric[1] * num_epochs / timer.sum():.1f}'
          f' examples/sec on {str(devices)}')
```

17.14.6 モデルの学習と検証

それではモデルを学習し、検証できる。以下のハイパーパラメータはすべて調整可能である。たとえば、エポック数は増やせる。lr_period と lr_decay はそれぞれ 2 と 0.9 に設定されているため、最適化アルゴリズムの学習率は 2 エポックごとに 0.9 倍になる。

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 10, 1e-4, 1e-4
lr_period, lr_decay, net = 2, 0.9, get_net(devices)
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

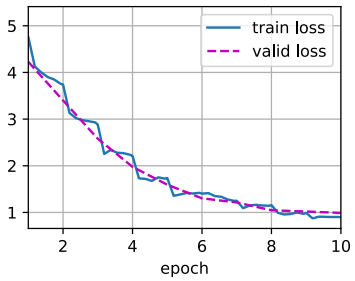
```
train loss 1.248, valid loss 1.476
619.4 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```




```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 10, 5e-3, 1e-4
lr_period, lr_decay, net = 2, 0.9, get_net(devices)
net.hybridize()
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

```
train loss 0.896, valid loss 0.990
268.6 examples/sec on [gpu(0), gpu(1)]
```

17.14.7 テストセットの分類と Kaggle への結果提出

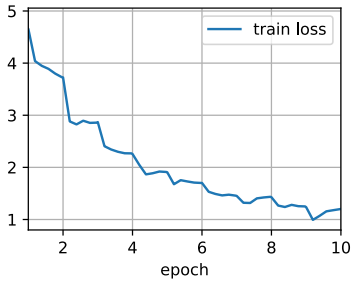


Section 17.13 の最後の手順と同様に、最終的には、ラベル付きデータ（検証セットを含む）すべてを使ってモデルを学習し、テストセットを分類する。分類には学習済みのカスタム出力ネットワークを使用する。

```
net = get_net(devices)
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

preds = []
for data, label in test_iter:
    output = torch.nn.functional.softmax(net(data.to(devices[0])), dim=1)
    preds.extend(output.cpu().detach().numpy())
ids = sorted(os.listdir(
    os.path.join(data_dir, 'train_valid_test', 'test', 'unknown')))
with open('submission.csv', 'w') as f:
    f.write('id,' + ','.join(train_valid_ds.classes) + '\n')
    for i, output in zip(ids, preds):
        f.write(i.split('.')[0] + ',' + ','.join(
            [str(num) for num in output]) + '\n')
```

```
train loss 1.203
744.5 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```




```
net = get_net(devices)
net.hybridize()
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
```

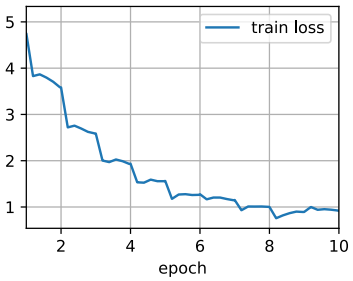
(continues on next page)

(continued from previous page)

```
      lr_decay)

preds = []
for data, label in test_iter:
    output_features = net.features(data.as_in_ctx(devices[0]))
    output = npx.softmax(net.output_new(output_features))
    preds.extend(output.asnumpy())
ids = sorted(os.listdir(
    os.path.join(data_dir, 'train_valid_test', 'test', 'unknown')))
with open('submission.csv', 'w') as f:
    f.write('id,' + ','.join(train_valid_ds.synsets) + '\n')
    for i, output in zip(ids, preds):
        f.write(i.split('.')[0] + ',' + ','.join(
            [str(num) for num in output]) + '\n')
```

```
train loss 0.914
296.1 examples/sec on [gpu(0), gpu(1)]
```



上のコードは `submission.csv` ファイルを生成し、Section 8.7 で説明したのと同じ方法で Kaggle に提出できる。

17.14.8 まとめ

- ImageNet データセットの画像は CIFAR-10 の画像よりも大きく、サイズもさまざまである。異なるデータセットのタスクでは、画像拡張操作を変更してよい場合がある。
- ImageNet データセットのサブセットを分類するには、完全な ImageNet データセットで事前学習されたモデルを利用して特徴を抽出し、カスタムの小規模出力ネットワークのみを学習すればよいである。これにより、計算時間とメモリ使用量を削減できる。

17.14.9 演習

1. Kaggle コンペティションの完全なデータセットを使う場合、`batch_size`（バッチサイズ）と `num_epochs`（エポック数）を増やし、他のハイパーパラメータを `lr = 0.01`, `lr_period = 10`, `lr_decay = 0.1` に設定すると、どのような結果が得られるか？

2. より深い事前学習済みモデルを使うと、より良い結果が得られるか？ハイパーパラメータはどのように調整するか？さらに結果を改善できるか？

18 自然言語処理: 事前学習

人間にはコミュニケーションが必要である。人間の本質的なこの必要性から、膨大な量の書き言葉が日々生成されている。ソーシャルメディア、チャットアプリ、メール、製品レビュー、ニュース記事、研究論文、書籍に豊富なテキストがあることを踏まえると、人間の言語に基づいて支援を提供したり意思決定を行ったりできるように、コンピュータがそれらを理解できるようにすることが極めて重要になる。

自然言語処理は、自然言語を用いたコンピュータと人間の相互作用を研究する。実際には、Section 12.3 の言語モデルや Section 13.5 の機械翻訳モデルのように、自然言語処理技術を用いてテキスト（人間の自然言語）データを処理・分析することが非常に一般的である。

テキストを理解するには、まずその表現を学習することから始められる。大規模コーパスに含まれる既存のテキスト系列を活用することで、自己教師あり学習は、テキストの周囲の別の部分を用いてテキストの一部の隠れた部分を予測するなどして、テキスト表現の事前学習に広く用いられてきた。このようにして、モデルは大規模なテキストデータから、高価なラベル付け作業なしに教師あり学習を通じて学習する！

この章で見るように、各単語やサブワードを個別のトークンとして扱うとき、各トークンの表現は、word2vec、GloVe、またはサブワード埋め込みモデルを用いて大規模コーパス上で事前学習できる。事前学習後、各トークンの表現はベクトルにできるが、文脈が何であっても同じままである。たとえば、“bank”のベクトル表現は、“go to the bank to deposit some money”と“go to the bank to sit down”の両方で同じである。そのため、より最近の多くの事前学習モデルでは、同じトークンの表現を異なる文脈に適応させる。その代表例が BERT であり、Transformer エンコーダに基づく、はるかに深い自己教師ありモデルである。この章では、Fig. 18.1 に示すように、テキストのためにこのような表現をどのように事前学習するかに焦点を当てる。

全体像を把握するために、Fig. 18.1 は、事前学習されたテキスト表現が、さまざまな下流の自然言語処理アプリケーションに対して、多様な深層学習アーキテクチャへ入力できることを示している。それらについては Section 19 で扱いる。

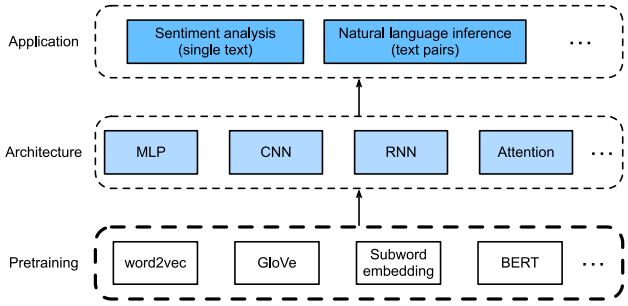


Fig. 18.1 事前学習されたテキスト表現は、さまざまな下流の自然言語処理アプリケーションに対して、各種の深層学習アーキテクチャへ入力できる。この章では、上流のテキスト表現の事前学習に焦点を当てる。

18.1 単語埋め込み（word2vec）

自然言語は、意味を表現するために用いられる複雑なシステムである。このシステムでは、単語が意味の基本単位である。その名のとおり、単語ベクトルは単語を表現するためのベクトルであり、単語の特徴ベクトル、あるいは表現とみなすこともできる。単語を実数ベクトルに写像する技術は 単語埋め込みと呼ばれる。近年、単語埋め込みは徐々に自然言語処理の基礎知識となってきた。

18.1.1 ワンホットベクトルは悪い選択

Section 12.5 では、単語（文字は単語）を表現するためにワンホットベクトルを用いた。辞書中の異なる単語の数（辞書サイズ）を N とし、各単語に 0 から $N-1$ までの異なる整数（インデックス）を対応させるとする。インデックスが i の任意の単語に対してワンホットベクトル表現を得るには、長さ N の全要素 0 のベクトルを作成し、位置 i の要素を 1 に設定する。このようにして、各単語は長さ N のベクトルで表現され、ニューラルネットワークで直接利用できる。

ワンホット単語ベクトルは簡単に構成できるが、通常は良い選択ではない。主な理由の 1 つは、ワンホット単語ベクトルでは、私たちがよく用いる コサイン類似度のように、異なる単語間の類似性を正確に表現できないことである。ベクトル $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ に対して、そのコサイン類似度は両者のなす角の余弦である。

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} \in [-1, 1]. \tag{18.1.1}$$

任意の異なる 2 つの単語のワンホットベクトル間のコサイン類似度は 0 なので、ワンホットベクトルは単語間の類似性を符号化できない。

18.1.2 自己教師あり word2vec

112


上記の問題に対処するために word2vec [112] ツールが提案された。これは各単語を固定長ベクトルに写像し、これらのベクトルは異なる単語間の類似性や類推関係をよりよく表現できる。word2vec ツールには 2 つのモデル、すなわち *skip-gram* (Mikolov *et al.*, 2013) と *continuous bag of words*（CBOW）(Mikolov *et al.*, 2013) がある。意味的に有意味な表現を得るために、その学習は条件付き確率に依存しており、コーパス中の周囲のいくつかの単語を使っていくつかの単語を予測するものとみなせる。教師信号がラベルなしデータから得られるため、skip-gram と continuous bag of words の両方は自己教師ありモデルである。

以下では、これら 2 つのモデルとその学習方法を紹介する。

18.1.3 Skip-Gram モデル

skip-gram モデルは、ある単語からテキスト系列中の周囲の単語を生成できると仮定する。テキスト系列“the”, “man”, “loves”, “his”, “son”を例に取りよう。“loves”を 中心単語とし、コンテキストウィンドウサイズを 2 に設定する。Fig. 18.1.1 に示すように、中心単語“loves”が与えられたとき、skip-gram モデルは、中心単語から 2 語以内にある 文脈単語“the”, “man”, “his”, “son”を生成する条件付き確率を考える。

$$P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} \mid \text{"loves"}). \tag{18.1.2}$$

中心単語が与えられたときに文脈単語が独立に生成される（すなわち条件付き独立）と仮定する。この場合、上の条件付き確率は次のように書き換えられる。

$$P(\text{"the"} \mid \text{"loves"}) \cdot P(\text{"man"} \mid \text{"loves"}) \cdot P(\text{"his"} \mid \text{"loves"}) \cdot P(\text{"son"} \mid \text{"loves"}). \tag{18.1.3}$$

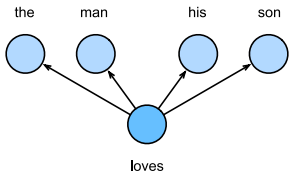


Fig. 18.1.1 The skip-gram model considers the conditional probability of generating the surrounding context words given a center word.

skip-gram モデルでは、各単語は条件付き確率を計算するために 2 つの d 次元ベクトル表現を持つ。より具体的には、辞書中のインデックスが i の任意の単語について、$\mathbf{v}_i \in \mathbb{R}^d$ および $\mathbf{u}_i \in \mathbb{R}^d$ を、それぞれ 中心単語および 文脈単語として用いるときの 2 つのベクトルとする。辞書中のインデックスが o の任意の文脈単語 w_o を、辞書中のインデックスが c の中心単語 w_c から生成する条件付き確率

は、ベクトルの内積に対する softmax 演算でモデル化できる。

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}, \tag{18.1.4}$$

ここで、語彙インデックス集合 $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$ である。長さ T のテキスト系列があり、時刻 t の単語を $w^{(t)}$ と表す。任意の中心単語が与えられたときに文脈単語が独立に生成されると仮定する。コンテキストウィンドウサイズを m とすると、skip-gram モデルの尤度関数は、任意の中心単語が与えられたときにすべての文脈単語を生成する確率である。

$$\prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \ne 0} P(w^{(t+j)} \mid w^{(t)}), \tag{18.1.5}$$

ここで、1 未満または T を超える時刻は省略できる。

学習

skip-gram モデルのパラメータは、語彙中の各単語に対する中心単語ベクトルと文脈単語ベクトルである。学習では、尤度関数を最大化する（すなわち最尤推定を行う）ことでモデルパラメータを学習する。これは次の損失関数を最小化することと等価である。

$$-\sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \ne 0} \log P(w^{(t+j)} \mid w^{(t)}). \tag{18.1.6}$$

確率的勾配降下法を用いて損失を最小化するとき、各反復でより短い部分系列をランダムにサンプリングし、その部分系列に対する（確率的）勾配を計算してモデルパラメータを更新できる。この（確率的）勾配を計算するには、中心単語ベクトルと文脈単語ベクトルに関する対数条件付き確率の勾配を求める必要がある。一般に、(18.1.4) に従うと、中心単語 w_c と文脈単語 w_o の任意の組に関わる対数条件付き確率は

$$\log P(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log\left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)\right). \tag{18.1.7}$$

微分により、その中心単語ベクトル $\mathbf{v}_c$ に関する勾配は次のように得られる。

$$\begin{aligned}\frac{\partial \log P(w_o \mid w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c)\mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \left(\frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}\right)\mathbf{u}_j \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_j \mid w_c)\mathbf{u}_j.\end{aligned} \tag{18.1.8}$$

(18.1.8) の計算には、中心単語として w_c を用いたときの辞書中のすべての単語の条件付き確率が必要であることに注意してほしい。他の単語ベクトルに対する勾配も同様に求められる。

学習後、辞書中のインデックスが i の任意の単語について、2 つの単語ベクトル $\mathbf{v}_i$（中心単語として）と $\mathbf{u}_i$（文脈単語として）を得る。自然言語処理の応用では、skip-gram モデルの中心単語ベクトルが通常単語表現として用いられる。

18.1.4 Continuous Bag of Words（CBOW）モデル

continuous bag of words（CBOW）モデルは skip-gram モデルに似ている。skip-gram モデルとの主な違いは、continuous bag of words モデルがテキスト系列中の周囲の文脈単語に基づいて中心単語が生成されると仮定する点である。たとえば、同じテキスト系列“the”, “man”, “loves”, “his”, “son”で、“loves”を中心単語とし、コンテキストウィンドウサイズを 2 とすると、continuous bag of words モデルは文脈単語“the”, “man”, “his”, “son”に基づいて中心単語“loves”を生成する条件付き確率を考えます（Fig. 18.1.2 を参照）。これは

$$P(\text{"loves"} \mid \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"}). \tag{18.1.9}$$

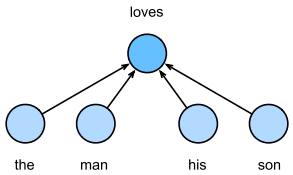


Fig. 18.1.2 The continuous bag of words model considers the conditional probability of generating the center word given its surrounding context words.

continuous bag of words モデルでは複数の文脈単語があるため、条件付き確率の計算ではこれらの文脈単語ベクトルを平均する。具体的には、辞書中のインデックスが i の任意の単語について、$\mathbf{v}_i \in \mathbb{R}^d$ および $\mathbf{u}_i \in \mathbb{R}^d$ を、それぞれ 文脈単語および 中心単語として用いるときの 2 つのベクトルとする（意味は skip-gram モデルと入れ替わっている）。周囲の文脈単語 $w_{o_1}, \ldots, w_{o_{2m}}$（辞書中のインデックスが $o_1, \ldots, o_{2m}$）が与えられたときに、任意の中心単語 w_c（辞書中のインデックスが c）を生成する条件付き確率は、次のようにモデル化できる。

$$P(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top (\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top (\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}. \tag{18.1.10}$$

簡潔のために、$\mathcal{W}_o = \{w_{o_1}, \ldots, w_{o_{2m}}\}$ および $\bar{\mathbf{v}}_o = \left(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}}\right)/(2m)$ とおきる。すると (18.1.10) は次のように簡略化できる。

$$P(w_c \mid \mathcal{W}_o) = \frac{\exp\left(\mathbf{u}_c^\top \bar{\mathbf{v}}_o\right)}{\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)}. \tag{18.1.11}$$

長さ T のテキスト系列があり、時刻 t の単語を $w^{(t)}$ と表す。コンテキストウィンドウサイズを m とすると、continuous bag of words モデルの尤度関数は、そ

れぞれの文脈単語が与えられたときにすべての中心単語を生成する確率である。

$$\prod_{t=1}^{T} P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{18.1.12}$$

学習

continuous bag of words モデルの学習は、skip-gram モデルの学習とほぼ同じである。continuous bag of words モデルの最尤推定は、次の損失関数を最小化することと等価である。

$$-\sum_{t=1}^{T} \log P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{18.1.13}$$

次に注意しよ。

$$\log\, P(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left(\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right) \right). \tag{18.1.14}$$

微分により、任意の文脈単語ベクトル $\mathbf{v}_{o_i}$（$i = 1, \ldots, 2m$）に関する勾配は次のように得られる。

$$\frac{\partial \log\, P(w_c \mid \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o)\mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)} \right) = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} P(w_j \mid \mathcal{W}_o)\mathbf{u}_j \right). \tag{18.1.15}$$

他の単語ベクトルに対する勾配も同様に求められる。skip-gram モデルとは異なり、continuous bag of words モデルでは通常文脈単語ベクトルを単語表現として用いる。

18.1.5 まとめ

- 単語ベクトルは単語を表現するためのベクトルであり、単語の特徴ベクトル、あるいは表現とみなすこともできる。単語を実数ベクトルに写像する技術は単語埋め込みと呼ばれる。
- word2vec ツールには skip-gram モデルと continuous bag of words モデルの両方が含まれる。
- skip-gram モデルは、ある単語からテキスト系列中の周囲の単語を生成できると仮定する。一方、continuous bag of words モデルは、周囲の文脈単語に基づいて中心単語が生成されると仮定する。

18.1.6 演習

1. 各勾配を計算する計算量はどれくらいですか？辞書サイズが非常に大きい場合、どのような問題が生じますか？

2. 英語のいくつかの定型句は“new york”のように複数の単語から構成される。これらの単語ベクトルをどのように学習するか？ヒント：word2vec 論文の Section 4 を参照されたい (Mikolov *et al.*, 2013)。

3. skip-gram モデルを例に、word2vec の設計について考えてみよう。skip-gram モデルにおける 2 つの単語ベクトルの内積とコサイン類似度の関係は何だろうか？意味的に似た単語の組に対して、なぜ skip-gram モデルで学習した単語ベクトルのコサイン類似度が高くなりうるのだろうか？

18.2 近似学習

Section 18.1 での議論を思い出してほしい。
skip-gram モデルの主な考え方は、softmax 演算を用いて、与えられた中心語 w_c に基づいて文脈語 w_o を生成する条件付き確率を (18.1.4) で計算することである。その対応する対数損失は (18.1.7) の符号を反転したものとして与えられる。

softmax 演算の性質上、文脈語は辞書 $\mathcal{V}$ のどれでもあり得るため、(18.1.7) の符号を反転した式には、語彙全体の大きさに等しい数の項の総和が含まれる。したがって、(18.1.8) における skip-gram モデルの勾配計算と、(18.1.15) における continuous bag-of-words モデルの勾配計算の両方に、この総和が含まれる。残念ながら、このように大きな辞書（しばしば数十万語から数百万語）にわたって和を取る勾配の計算コストは非常に大きいのです！

前述の計算複雑性を削減するために、この節では 2 つの近似学習法、*negative sampling* と *hierarchical softmax* を紹介する。skip-gram モデルと continuous bag of words モデルの類似性のため、ここでは skip-gram モデルを例として、これら 2 つの近似学習法を説明する。

18.2.1 Negative Sampling

Negative sampling は元の目的関数を修正する。中心語 w_c の文脈ウィンドウが与えられたとき、任意の（文脈）語 w_o がこの文脈ウィンドウから来るという事実を、次のようにモデル化される確率をもつ事象として考える。

$$P(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c), \tag{18.2.1}$$

ここで σ はシグモイド活性化関数を次のように定義したものである。

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{18.2.2}$$

まず、単語埋め込みを学習するために、テキスト系列におけるこのような事象すべての同時確率を最大化することから始めよう。具体的には、長さ T のテキ

スト系列が与えられたとき、時刻 t における単語を $w^{(t)}$ と表し、文脈ウィンドウサイズを m とすると、次の同時確率を最大化することを考える。

$$\prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \ne 0} P(D = 1 \mid w^{(t)}, w^{(t+j)}). \tag{18.2.3}$$

しかし、(18.2.3) は正例に関わる事象しか考慮していない。その結果、(18.2.3) の同時確率は、すべての単語ベクトルが無限大に等しい場合にのみ 1 に最大化される。もちろん、このような結果には意味がない。目的関数をより意味のあるものにするために、*negative sampling* はあらかじめ定めた分布からサンプリングした負例を追加する。

S を、文脈語 w_o が中心語 w_c の文脈ウィンドウから来るという事象とする。この w_o を含む事象に対して、あらかじめ定めた分布 $P(w)$ から、この文脈ウィンドウに含まれない K 個の *noise words* をサンプルする。N_k を、ノイズ語 w_k $(k = 1, \ldots, K)$ が w_c の文脈ウィンドウから来ないという事象とする。正例と負例の両方に関わるこれらの事象 $S, N_1, \ldots, N_K$ は互いに独立であると仮定する。Negative sampling は、(18.2.3) における（正例のみを含む）同時確率を次のように書き換える。

$$\prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \ne 0} P(w^{(t+j)} \mid w^{(t)}), \tag{18.2.4}$$

ここで条件付き確率は、事象 $S, N_1, \ldots, N_K$ を通じて近似される。

$$P(w^{(t+j)} \mid w^{(t)}) = P(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1,\ w_k \sim P(w)}^{K} P(D = 0 \mid w^{(t)}, w_k). \tag{18.2.5}$$

テキスト系列の時刻 t における単語 $w^{(t)}$ のインデックスを i_t、ノイズ語 w_k のインデックスを h_k とする。(18.2.5) における条件付き確率に関する対数損失は次のようになる。

$$\begin{aligned}
-\log P(w^{(t+j)} \mid w^{(t)}) &= -\log P(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log P(D = 0 \mid w^{(t)}, w_k) \\
&= -\log \sigma\left(\mathbf{u}_{i_{t+j}}^{\top} \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log\left(1 - \sigma\left(\mathbf{u}_{h_k}^{\top} \mathbf{v}_{i_t}\right)\right) \\
&= -\log \sigma\left(\mathbf{u}_{i_{t+j}}^{\top} \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log \sigma\left(-\mathbf{u}_{h_k}^{\top} \mathbf{v}_{i_t}\right).
\end{aligned} \tag{18.2.6}$$

これにより、各学習ステップでの勾配計算コストは辞書サイズに依存せず、K に線形に依存することがわかる。ハイパーパラメータ K を小さく設定すれば、negative sampling を用いた各学習ステップでの勾配計算コストはより小さくなる。

18.2.2 Hierarchical Softmax

別の近似学習法として、*hierarchical softmax* は二分木を用いる。これは Fig. 18.2.1 に示すデータ構造で、木の各葉ノードが辞書 $\mathcal{V}$ の単語を表す。

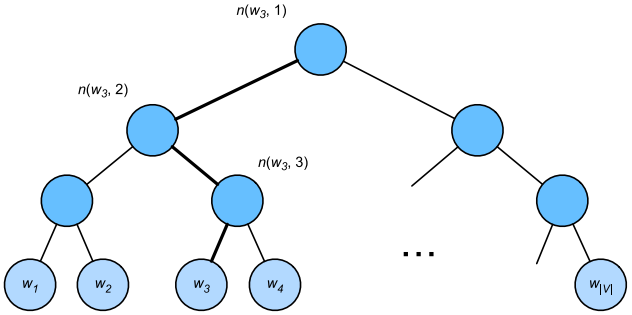


Fig. 18.2.1 近似学習のための hierarchical softmax。各葉ノードは辞書中の単語を表す。

$L(w)$ を、二分木において根ノードから単語 w を表す葉ノードまでの経路上にあるノード数（両端を含む）とする。$n(w, j)$ をこの経路上の j^{th} ノードとし、その文脈語ベクトルを $\mathbf{u}_{n(w,j)}$ とする。たとえば、Fig. 18.2.1 では $L(w_3) = 4$ である。Hierarchical softmax は、(18.1.4) における条件付き確率を次のように近似する。

$$P(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma\left([\![n(w_o, j+1) = \text{leftChild}(n(w_o, j))]\!] \cdot \mathbf{u}_{n(w_o,j)}^\top \mathbf{v}_c\right), \tag{18.2.7}$$

ここで関数 σ は (18.2.2) で定義され、leftChild(n) はノード n の左の子ノードである。x が真なら $[\![x]\!] = 1$、そうでなければ $[\![x]\!] = -1$ である。

例として、Fig. 18.2.1 において、単語 w_c が与えられたときに単語 w_3 を生成する条件付き確率を計算してみよう。これには、w_c の単語ベクトル $\mathbf{v}_c$ と、根から w_3 までの経路上にある非葉ノードのベクトル（Fig. 18.2.1 で太字で示された経路）との内積が必要である。この経路は左、右、左の順にたどられる。

$$P(w_3 \mid w_c) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_c). \tag{18.2.8}$$

$\sigma(x) + \sigma(-x) = 1$ なので、任意の単語 w_c に基づいて辞書 $\mathcal{V}$ のすべての単語を生成する条件付き確率の総和は 1 になる。

$$\sum_{w \in \mathcal{V}} P(w \mid w_c) = 1. \tag{18.2.9}$$

幸いなことに、二分木構造のため $L(w_o) - 1$ は $O(\log_2|\mathcal{V}|)$ のオーダーである。したがって、辞書サイズ $\mathcal{V}$ が非常に大きい場合、hierarchical softmax を用いた各学習ステップの計算コストは、近似学習を用いない場合と比べて大幅に削減される。

18.2.3 まとめ

- Negative sampling は、正例と負例の両方に関わる互いに独立な事象を考慮して損失関数を構成する。学習時の計算コストは、各ステップでのノイズ語の数に線形に依存する。
- Hierarchical softmax は、二分木における根ノードから葉ノードまでの経路を用いて損失関数を構成する。学習時の計算コストは、各ステップでの辞書サイズの対数に依存する。

18.2.4 演習

1. Negative sampling では、どのようにノイズ語をサンプリングできるか？
2. (18.2.9) が成り立つことを確認しなさい。
3. continuous bag of words モデルを、それぞれ negative sampling と hierarchical softmax を用いてどのように学習するか？

18.3 単語埋め込みの事前学習のためのデータセット

word2vec モデルと近似学習手法の技術的な詳細が分かったので、それらの実装を見ていきよう。具体的には、Section 18.1 の skip-gram モデルと: numref:sec_approx_train の負例サンプリングを例にする。この節では、単語埋め込みモデルの事前学習のためのデータセットから始める。元の形式のデータは、学習中に反復して取り出せるミニバッチへと変換される。

```
import collections
from d2l import torch as d2l
import math
import torch
import os
import random
```

```
import collections
from d2l import mxnet as d2l
import math
from mxnet import gluon, np
import os
import random
```

```
#@save
d2l.DATA_HUB['ptb'] = (d2l.DATA_URL + 'ptb.zip',
```

(continues on next page)

(continued from previous page)

```
                      '319d85e578af0cdc590547f26231e4e31cdf1e42')

#@save
def read_ptb():
    """Load the PTB dataset into a list of text lines."""
    data_dir = d2l.download_extract('ptb')
    # Read the training set
    with open(os.path.join(data_dir, 'ptb.train.txt')) as f:
        raw_text = f.read()
    return [line.split() for line in raw_text.split('\n')]

sentences = read_ptb()
f'# sentences: {len(sentences)}'
```

```
#@save
d2l.DATA_HUB['ptb'] = (d2l.DATA_URL + 'ptb.zip',
                       '319d85e578af0cdc590547f26231e4e31cdf1e42')

#@save
def read_ptb():
    """Load the PTB dataset into a list of text lines."""
    data_dir = d2l.download_extract('ptb')
    # Read the training set
    with open(os.path.join(data_dir, 'ptb.train.txt')) as f:
        raw_text = f.read()
    return [line.split() for line in raw_text.split('\n')]

sentences = read_ptb()
f'# sentences: {len(sentences)}'
```

18.3.1 データセットの読み込み

ここで使用するデータセットは Penn Tree Bank (PTB)[113] である。このコーパスは Wall Street Journal の記事からサンプリングされ、訓練、検証、テストの各セットに分割されている。元の形式では、テキストファイルの各行が空白で区切られた単語列からなる 1 つの文を表す。ここでは各単語をトークンとして扱いる。

113



```
#@save
d2l.DATA_HUB['ptb'] = (d2l.DATA_URL + 'ptb.zip',
                       '319d85e578af0cdc590547f26231e4e31cdf1e42')

#@save
def read_ptb():
    """Load the PTB dataset into a list of text lines."""
    data_dir = d2l.download_extract('ptb')
    # Read the training set
    with open(os.path.join(data_dir, 'ptb.train.txt')) as f:
        raw_text = f.read()
    return [line.split() for line in raw_text.split('\n')]
```

(continues on next page)

(continued from previous page)

```
sentences = read_ptb()
f'# sentences: {len(sentences)}'
```

```
Downloading ../data/ptb.zip from http://d2l-data.s3-accelerate.amazonaws.com/
↪ptb.zip...
```

```
'# sentences: 42069'
```

```
#@save
d2l.DATA_HUB['ptb'] = (d2l.DATA_URL + 'ptb.zip',
                       '319d85e578af0cdc590547f26231e4e31cdf1e42')

#@save
def read_ptb():
    """Load the PTB dataset into a list of text lines."""
    data_dir = d2l.download_extract('ptb')
    # Read the training set
    with open(os.path.join(data_dir, 'ptb.train.txt')) as f:
        raw_text = f.read()
    return [line.split() for line in raw_text.split('\n')]

sentences = read_ptb()
f'# sentences: {len(sentences)}'
```

```
Downloading ../data/ptb.zip from http://d2l-data.s3-accelerate.amazonaws.com/
↪ptb.zip...
```

```
'# sentences: 42069'
```

```
vocab = d2l.Vocab(sentences, min_freq=10)
f'vocab size: {len(vocab)}'
```

```
vocab = d2l.Vocab(sentences, min_freq=10)
f'vocab size: {len(vocab)}'
```

訓練セットを読み込んだ後、コーパスの語彙を構築する。その際、10 回未満しか現れない単語はすべて“<unk>”トークンに置き換えられる。元のデータセットにも、まれな（未知の）単語を表す“<unk>”トークンが含まれていることに注意しよ。

```
vocab = d2l.Vocab(sentences, min_freq=10)
f'vocab size: {len(vocab)}'
```

```
'vocab size: 6719'
```

```
vocab = d2l.Vocab(sentences, min_freq=10)
f'vocab size: {len(vocab)}'
```

```
'vocab size: 6719'
```

```
#@save
def subsample(sentences, vocab):
    """Subsample high-frequency words."""
    # Exclude unknown tokens ('<unk>')
    sentences = [[token for token in line if vocab[token] != vocab.unk]
                 for line in sentences]
    counter = collections.Counter([
        token for line in sentences for token in line])
    num_tokens = sum(counter.values())

    # Return True if `token` is kept during subsampling
    def keep(token):
        return(random.uniform(0, 1) <
               math.sqrt(1e-4 / counter[token] * num_tokens))

    return ([[token for token in line if keep(token)] for line in sentences],
            counter)

subsampled, counter = subsample(sentences, vocab)
```

```
#@save
def subsample(sentences, vocab):
    """Subsample high-frequency words."""
    # Exclude unknown tokens ('<unk>')
    sentences = [[token for token in line if vocab[token] != vocab.unk]
                 for line in sentences]
    counter = collections.Counter([
        token for line in sentences for token in line])
    num_tokens = sum(counter.values())

    # Return True if `token` is kept during subsampling
    def keep(token):
        return(random.uniform(0, 1) <
               math.sqrt(1e-4 / counter[token] * num_tokens))

    return ([[token for token in line if keep(token)] for line in sentences],
            counter)

subsampled, counter = subsample(sentences, vocab)
```

18.3.2 サブサンプリング

テキストデータには通常、“the”、“a”、“in”のような高頻度語が含まれる。非常に大きなコーパスでは、これらの語は数十億回も出現することがある。しかし、これらの語は文脈ウィンドウ内で多くの異なる単語と共起するため、有用な信号はあまり提供しない。たとえば、文脈ウィンドウ内の単語“chip”を考えてみよう。直感的には、低頻度語“intel”との共起のほうが、高頻度語“a”との共起よりも学習において有用である。さらに、（高頻度の）単語を大量に用いて学習するのは遅くなる。したがって、単語埋め込みモデルを学習する際には、高頻度語を サブサンプリングする (Mikolov *et al.*, 2013)。具体的には、データセット中の各インデックス付き単語 w_i は、次の確率で破棄される。

$$P(w_i) = \max\left(1 - \sqrt{\frac{t}{f(w_i)}}, 0\right), \tag{18.3.1}$$

ここで $f(w_i)$ は、単語 w_i の出現数をデータセット中の総単語数で割った比率であり、定数 t はハイパーパラメータです（実験では 10^{-4}）。相対頻度 $f(w_i) > t$ のときにのみ（高頻度の）単語 w_i が破棄されうることが分かりる。また、単語の相対頻度が高いほど、破棄される確率も高くなる。

```
#@save
def subsample(sentences, vocab):
    """Subsample high-frequency words."""
    # Exclude unknown tokens ('<unk>')
    sentences = [[token for token in line if vocab[token] != vocab.unk]
                 for line in sentences]
    counter = collections.Counter([
        token for line in sentences for token in line])
    num_tokens = sum(counter.values())

    # Return True if `token` is kept during subsampling
    def keep(token):
        return(random.uniform(0, 1) <
               math.sqrt(1e-4 / counter[token] * num_tokens))

    return ([[token for token in line if keep(token)] for line in sentences],
            counter)

subsampled, counter = subsample(sentences, vocab)
```

```
#@save
def subsample(sentences, vocab):
    """Subsample high-frequency words."""
    # Exclude unknown tokens ('<unk>')
    sentences = [[token for token in line if vocab[token] != vocab.unk]
                 for line in sentences]
    counter = collections.Counter([
        token for line in sentences for token in line])
    num_tokens = sum(counter.values())

    # Return True if `token` is kept during subsampling
    def keep(token):
        return(random.uniform(0, 1) <
```

(continues on next page)

(continued from previous page)

```
                math.sqrt(1e-4 / counter[token] * num_tokens))

    return ([[token for token in line if keep(token)] for line in sentences],
            counter)

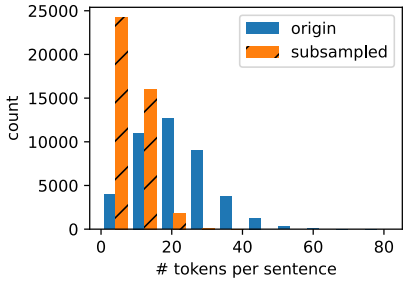
subsampled, counter = subsample(sentences, vocab)
```

```
d2l.show_list_len_pair_hist(['origin', 'subsampled'], '# tokens per sentence',
                            'count', sentences, subsampled);
```

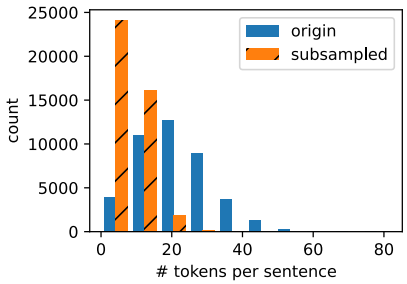

```
d2l.show_list_len_pair_hist(['origin', 'subsampled'], '# tokens per sentence',
                            'count', sentences, subsampled);
```


次のコード片は、サブサンプリング前後の 1 文あたりのトークン数のヒストグラムを描画する。予想どおり、サブサンプリングによって高頻度語が削除され、文が大幅に短くなっており、これにより学習が高速化される。


```
d2l.show_list_len_pair_hist(['origin', 'subsampled'], '# tokens per sentence',
                            'count', sentences, subsampled);
```


```
d2l.show_list_len_pair_hist(['origin', 'subsampled'], '# tokens per sentence',
                            'count', sentences, subsampled);
```

```
def compare_counts(token):
    return (f'# of "{token}": '
            f'before={sum([l.count(token) for l in sentences])}, '
            f'after={sum([l.count(token) for l in subsampled])}')

compare_counts('the')
```

```
def compare_counts(token):
    return (f'# of "{token}": '
            f'before={sum([l.count(token) for l in sentences])}, '
            f'after={sum([l.count(token) for l in subsampled])}')

compare_counts('the')
```

個々のトークンについて見ると、高頻度語“the”のサンプリング率は 1/20 未満である。

```
def compare_counts(token):
    return (f'# of "{token}": '
            f'before={sum([l.count(token) for l in sentences])}, '
            f'after={sum([l.count(token) for l in subsampled])}')

compare_counts('the')
```

```
'# of "the": before=50770, after=1929'
```

```
def compare_counts(token):
    return (f'# of "{token}": '
            f'before={sum([l.count(token) for l in sentences])}, '
            f'after={sum([l.count(token) for l in subsampled])}')

compare_counts('the')
```

```
'# of "the": before=50770, after=2004'
```

```
compare_counts('join')
```

```
compare_counts('join')
```

対照的に、低頻度語“join”は完全に保持される。

```
compare_counts('join')
```

```
'# of "join": before=45, after=45'
```

```
compare_counts('join')
```

```
'# of "join": before=45, after=45'
```

```
corpus = [vocab[line] for line in subsampled]
corpus[:3]
```

```
corpus = [vocab[line] for line in subsampled]
corpus[:3]
```

サブサンプリング後、コーパスのトークンをインデックスに変換する。

```
corpus = [vocab[line] for line in subsampled]
corpus[:3]
```

```
[[], [6697, 3228, 1773], [3922, 1922, 4743]]
```

```
corpus = [vocab[line] for line in subsampled]
corpus[:3]
```

```
[[], [4127, 3228, 710], [3895, 993, 3922, 1922, 4743]]
```

```
#@save
def get_centers_and_contexts(corpus, max_window_size):
    """Return center words and context words in skip-gram."""
    centers, contexts = [], []
    for line in corpus:
        # To form a "center word--context word" pair, each sentence needs to
        # have at least 2 words
        if len(line) < 2:
            continue
        centers += line
        for i in range(len(line)):  # Context window centered at `i`
            window_size = random.randint(1, max_window_size)
            indices = list(range(max(0, i - window_size),
                                 min(len(line), i + 1 + window_size)))
            # Exclude the center word from the context words
            indices.remove(i)
            contexts.append([line[idx] for idx in indices])
    return centers, contexts
```

```
#@save
def get_centers_and_contexts(corpus, max_window_size):
    """Return center words and context words in skip-gram."""
    centers, contexts = [], []
```

(continues on next page)

(continued from previous page)

```
    for line in corpus:
        # To form a "center word--context word" pair, each sentence needs to
        # have at least 2 words
        if len(line) < 2:
            continue
        centers += line
        for i in range(len(line)):  # Context window centered at `i`
            window_size = random.randint(1, max_window_size)
            indices = list(range(max(0, i - window_size),
                                 min(len(line), i + 1 + window_size)))
            # Exclude the center word from the context words
            indices.remove(i)
            contexts.append([line[idx] for idx in indices])
    return centers, contexts
```

18.3.3 中心語と文脈語の抽出

次の get_centers_and_contexts 関数は、corpus からすべての中心語とその文脈語を抽出す。文脈ウィンドウサイズとして、1 から max_window_size までの整数を一様にランダムサンプリングする。任意の中心語について、その単語からの距離がサンプリングされた文脈ウィンドウサイズを超えない単語が文脈語になる。

```
#@save
def get_centers_and_contexts(corpus, max_window_size):
    """Return center words and context words in skip-gram."""
    centers, contexts = [], []
    for line in corpus:
        # To form a "center word--context word" pair, each sentence needs to
        # have at least 2 words
        if len(line) < 2:
            continue
        centers += line
        for i in range(len(line)):  # Context window centered at `i`
            window_size = random.randint(1, max_window_size)
            indices = list(range(max(0, i - window_size),
                                 min(len(line), i + 1 + window_size)))
            # Exclude the center word from the context words
            indices.remove(i)
            contexts.append([line[idx] for idx in indices])
    return centers, contexts
```

```
#@save
def get_centers_and_contexts(corpus, max_window_size):
    """Return center words and context words in skip-gram."""
    centers, contexts = [], []
    for line in corpus:
        # To form a "center word--context word" pair, each sentence needs to
        # have at least 2 words
        if len(line) < 2:
            continue
```

(continues on next page)

(continued from previous page)

```
        centers += line
        for i in range(len(line)):  # Context window centered at `i`
            window_size = random.randint(1, max_window_size)
            indices = list(range(max(0, i - window_size),
                                 min(len(line), i + 1 + window_size)))
            # Exclude the center word from the context words
            indices.remove(i)
            contexts.append([line[idx] for idx in indices])
    return centers, contexts
```

```
tiny_dataset = [list(range(7)), list(range(7, 10))]
print('dataset', tiny_dataset)
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
    print('center', center, 'has contexts', context)
```


```
tiny_dataset = [list(range(7)), list(range(7, 10))]
print('dataset', tiny_dataset)
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
    print('center', center, 'has contexts', context)
```


次に、7 語と 3 語からなる 2 つの文を含む人工データセットを作成する。最大文脈ウィンドウサイズを 2 とし、すべての中心語とその文脈語を出力してみよう。


```
tiny_dataset = [list(range(7)), list(range(7, 10))]
print('dataset', tiny_dataset)
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
    print('center', center, 'has contexts', context)
```


```
dataset [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]
center 0 has contexts [1, 2]
center 1 has contexts [0, 2, 3]
center 2 has contexts [1, 3]
center 3 has contexts [2, 4]
center 4 has contexts [2, 3, 5, 6]
center 5 has contexts [4, 6]
center 6 has contexts [5]
center 7 has contexts [8, 9]
center 8 has contexts [7, 9]
center 9 has contexts [8]
```


```
tiny_dataset = [list(range(7)), list(range(7, 10))]
print('dataset', tiny_dataset)
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
    print('center', center, 'has contexts', context)
```


```
dataset [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]
```

(continues on next page)

(continued from previous page)

```
center 0 has contexts [1]
center 1 has contexts [0, 2]
center 2 has contexts [0, 1, 3, 4]
center 3 has contexts [1, 2, 4, 5]
center 4 has contexts [2, 3, 5, 6]
center 5 has contexts [4, 6]
center 6 has contexts [4, 5]
center 7 has contexts [8]
center 8 has contexts [7, 9]
center 9 has contexts [8]
```

```
all_centers, all_contexts = get_centers_and_contexts(corpus, 5)
f'# center-context pairs: {sum([len(contexts) for contexts in all_contexts])}'
```


```
all_centers, all_contexts = get_centers_and_contexts(corpus, 5)
f'# center-context pairs: {sum([len(contexts) for contexts in all_contexts])}'
```


PTB データセットで学習する際には、最大文脈ウィンドウサイズを 5 に設定する。次のコードは、データセット中のすべての中心語とその文脈語を抽出す。

```
all_centers, all_contexts = get_centers_and_contexts(corpus, 5)
f'# center-context pairs: {sum([len(contexts) for contexts in all_contexts])}'
```

```
'# center-context pairs: 1495467'
```

```
all_centers, all_contexts = get_centers_and_contexts(corpus, 5)
f'# center-context pairs: {sum([len(contexts) for contexts in all_contexts])}'
```

```
'# center-context pairs: 1500813'
```

```
#@save
class RandomGenerator:
    """Randomly draw among {1, ..., n} according to n sampling weights."""
    def __init__(self, sampling_weights):
        # Exclude
        self.population = list(range(1, len(sampling_weights) + 1))
        self.sampling_weights = sampling_weights
        self.candidates = []
        self.i = 0

    def draw(self):
        if self.i == len(self.candidates):
            # Cache `k` random sampling results
            self.candidates = random.choices(
                self.population, self.sampling_weights, k=10000)
```

(continues on next page)

(continued from previous page)

```
            self.i = 0
        self.i += 1
        return self.candidates[self.i - 1]
```

```
#@save
class RandomGenerator:
    """Randomly draw among {1, ..., n} according to n sampling weights."""
    def __init__(self, sampling_weights):
        # Exclude
        self.population = list(range(1, len(sampling_weights) + 1))
        self.sampling_weights = sampling_weights
        self.candidates = []
        self.i = 0

    def draw(self):
        if self.i == len(self.candidates):
            # Cache `k` random sampling results
            self.candidates = random.choices(
                self.population, self.sampling_weights, k=10000)
            self.i = 0
        self.i += 1
        return self.candidates[self.i - 1]
```

18.3.4 負例サンプリング

近似学習には負例サンプリングを用いる。あらかじめ定義された分布に従ってノイズ語をサンプリングするために、次の RandomGenerator クラスを定義する。ここで、（正規化されていない場合もある）サンプリング分布は引数 sampling_weights を通して渡される。

```
#@save
class RandomGenerator:
    """Randomly draw among {1, ..., n} according to n sampling weights."""
    def __init__(self, sampling_weights):
        # Exclude
        self.population = list(range(1, len(sampling_weights) + 1))
        self.sampling_weights = sampling_weights
        self.candidates = []
        self.i = 0

    def draw(self):
        if self.i == len(self.candidates):
            # Cache `k` random sampling results
            self.candidates = random.choices(
                self.population, self.sampling_weights, k=10000)
            self.i = 0
        self.i += 1
        return self.candidates[self.i - 1]
```

```
#@save
```

(continues on next page)

(continued from previous page)

```
class RandomGenerator:
    """Randomly draw among {1, ..., n} according to n sampling weights."""
    def __init__(self, sampling_weights):
        # Exclude
        self.population = list(range(1, len(sampling_weights) + 1))
        self.sampling_weights = sampling_weights
        self.candidates = []
        self.i = 0

    def draw(self):
        if self.i == len(self.candidates):
            # Cache `k` random sampling results
            self.candidates = random.choices(
                self.population, self.sampling_weights, k=10000)
            self.i = 0
        self.i += 1
        return self.candidates[self.i - 1]
```

```
#@save
def get_negatives(all_contexts, vocab, counter, K):
    """Return noise words in negative sampling."""
    # Sampling weights for words with indices 1, 2, ... (index 0 is the
    # excluded unknown token) in the vocabulary
    sampling_weights = [counter[vocab.to_tokens(i)]**0.75
                        for i in range(1, len(vocab))]
    all_negatives, generator = [], RandomGenerator(sampling_weights)
    for contexts in all_contexts:
        negatives = []
        while len(negatives) < len(contexts) * K:
            neg = generator.draw()
            # Noise words cannot be context words
            if neg not in contexts:
                negatives.append(neg)
        all_negatives.append(negatives)
    return all_negatives

all_negatives = get_negatives(all_contexts, vocab, counter, 5)
```

```
#@save
def get_negatives(all_contexts, vocab, counter, K):
    """Return noise words in negative sampling."""
    # Sampling weights for words with indices 1, 2, ... (index 0 is the
    # excluded unknown token) in the vocabulary
    sampling_weights = [counter[vocab.to_tokens(i)]**0.75
                        for i in range(1, len(vocab))]
    all_negatives, generator = [], RandomGenerator(sampling_weights)
    for contexts in all_contexts:
        negatives = []
        while len(negatives) < len(contexts) * K:
            neg = generator.draw()
            # Noise words cannot be context words
            if neg not in contexts:
                negatives.append(neg)
```

(continues on next page)

(continued from previous page)

```
        all_negatives.append(negatives)
    return all_negatives

all_negatives = get_negatives(all_contexts, vocab, counter, 5)
```

たとえば、サンプリング確率 $P(X = 1) = 2/9, P(X = 2) = 3/9, P(X = 3) = 4/9$ をもつインデックス 1, 2, 3 の中から 10 個の確率変数 X を次のようにサンプリングできる。

中心語と文脈語の組に対して、K 個（実験では 5 個）のノイズ語をランダムにサンプリングする。word2vec 論文の提案に従い、ノイズ語 w のサンプリング確率 $P(w)$ は、辞書内での相対頻度を 0.75 乗したものに設定する (Mikolov *et al.*, 2013)。

```
#@save
def get_negatives(all_contexts, vocab, counter, K):
    """Return noise words in negative sampling."""
    # Sampling weights for words with indices 1, 2, ... (index 0 is the
    # excluded unknown token) in the vocabulary
    sampling_weights = [counter[vocab.to_tokens(i)]**0.75
                        for i in range(1, len(vocab))]
    all_negatives, generator = [], RandomGenerator(sampling_weights)
    for contexts in all_contexts:
        negatives = []
        while len(negatives) < len(contexts) * K:
            neg = generator.draw()
            # Noise words cannot be context words
            if neg not in contexts:
                negatives.append(neg)
        all_negatives.append(negatives)
    return all_negatives

all_negatives = get_negatives(all_contexts, vocab, counter, 5)
```

```
generator = RandomGenerator([2, 3, 4])
[generator.draw() for _ in range(10)]
```

```
[2, 3, 3, 2, 1, 2, 3, 3, 3, 2]
```

```
#@save
def batchify(data):
    """Return a minibatch of examples for skip-gram with negative sampling."""
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]
```

(continues on next page)

(continued from previous page)

```
        labels += [[1] * len(context) + [0] * (max_len - len(context))]
    return (d2l.reshape(d2l.tensor(centers), (-1, 1)), d2l.tensor(
        contexts_negatives), d2l.tensor(masks), d2l.tensor(labels))
```

```
#@save
def batchify(data):
    """Return a minibatch of examples for skip-gram with negative sampling."""
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]
    return (d2l.reshape(d2l.tensor(centers), (-1, 1)), d2l.tensor(
        contexts_negatives), d2l.tensor(masks), d2l.tensor(labels))
```

18.3.5 ミニバッチで訓練例を読み込む

中心語、その文脈語、およびサンプリングされたノイズ語をすべて抽出した後、それらは学習中に反復的に読み込める訓練例のミニバッチへと変換される。

ミニバッチでは、i^{th} 個目の例に中心語と、n_i 個の文脈語および m_i 個のノイズ語が含まれる。文脈ウィンドウサイズが可変であるため、$n_i + m_i$ は i によって異なる。したがって、各例について文脈語とノイズ語を contexts_negatives 変数に連結し、連結長が $\max_i n_i + m_i$ (max_len) に達するまで 0 でパディングする。損失計算からパディングを除外するために、masks 変数を定義する。masks の要素と contexts_negatives の要素の間には 1 対 1 の対応があり、masks の 0（それ以外は 1）は contexts_negatives のパディングに対応する。

正例と負例を区別するために、labels 変数を用いて contexts_negatives 内の文脈語とノイズ語を分離する。masks と同様に、labels の要素と contexts_negatives の要素の間にも 1 対 1 の対応があり、labels の 1（それ以外は 0）は contexts_negatives 内の文脈語（正例）に対応する。

上の考え方は、次の batchify 関数で実装されている。その入力 data はバッチサイズと同じ長さのリストであり、各要素は中心語 center、その文脈語 context、およびそのノイズ語 negative からなる 1 つの例である。この関数は、マスク変数を含むなど、学習中の計算に読み込めるミニバッチを返す。

```
#@save
def batchify(data):
    """Return a minibatch of examples for skip-gram with negative sampling."""
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
```

(continues on next page)

(continued from previous page)

```
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]
    return (d2l.reshape(d2l.tensor(centers), (-1, 1)), d2l.tensor(
        contexts_negatives), d2l.tensor(masks), d2l.tensor(labels))
```

```
#@save
def get_negatives(all_contexts, vocab, counter, K):
    """Return noise words in negative sampling."""
    # Sampling weights for words with indices 1, 2, ... (index 0 is the
    # excluded unknown token) in the vocabulary
    sampling_weights = [counter[vocab.to_tokens(i)]**0.75
                        for i in range(1, len(vocab))]
    all_negatives, generator = [], RandomGenerator(sampling_weights)
    for contexts in all_contexts:
        negatives = []
        while len(negatives) < len(contexts) * K:
            neg = generator.draw()
            # Noise words cannot be context words
            if neg not in contexts:
                negatives.append(neg)
        all_negatives.append(negatives)
    return all_negatives

all_negatives = get_negatives(all_contexts, vocab, counter, 5)
```

```
x_1 = (1, [2, 2], [3, 3, 3, 3])
x_2 = (1, [2, 2, 2], [3, 3])
batch = batchify((x_1, x_2))

names = ['centers', 'contexts_negatives', 'masks', 'labels']
for name, data in zip(names, batch):
    print(name, '=', data)
```

```
x_1 = (1, [2, 2], [3, 3, 3, 3])
x_2 = (1, [2, 2, 2], [3, 3])
batch = batchify((x_1, x_2))

names = ['centers', 'contexts_negatives', 'masks', 'labels']
for name, data in zip(names, batch):
    print(name, '=', data)
```

2 つの例からなるミニバッチを使ってこの関数をテストしてみよう。

```
x_1 = (1, [2, 2], [3, 3, 3, 3])
x_2 = (1, [2, 2, 2], [3, 3])
batch = batchify((x_1, x_2))
```

(continues on next page)

(continued from previous page)

```
names = ['centers', 'contexts_negatives', 'masks', 'labels']
for name, data in zip(names, batch):
    print(name, '=', data)
```

```
centers = tensor([[1],
        [1]])
contexts_negatives = tensor([[2, 2, 3, 3, 3, 3],
        [2, 2, 2, 3, 3, 0]])
masks = tensor([[1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 0]])
labels = tensor([[1, 1, 0, 0, 0, 0],
        [1, 1, 1, 0, 0, 0]])
```

```
#@save
def batchify(data):
    """Return a minibatch of examples for skip-gram with negative sampling."""
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]
    return (d2l.reshape(d2l.tensor(centers), (-1, 1)), d2l.tensor(
        contexts_negatives), d2l.tensor(masks), d2l.tensor(labels))
```

18.3.6 まとめ

- 高頻度語は学習にあまり有用でない場合がある。学習を高速化するために、それらをサブサンプリングできる。
- 計算効率のため、例はミニバッチで読み込みる。パディングと非パディングを区別し、さらに正例と負例を区別するための変数を定義できる。

18.3.7 演習

1. サブサンプリングを使わない場合、この節のコードの実行時間はどのように変化するか？
2. RandomGenerator クラスは k 個の乱数サンプリング結果をキャッシュする。k を他の値に設定すると、データ読み込み速度にどのような影響があるか？
3. この節のコードにおける他のどのハイパーパラメータがデータ読み込み速度に影響しうるだろうか？

18.4 word2vec の事前学習

ここでは、Section 18.1 で定義した skip-gram モデルを実装する。その後、PTB データセット上で負例サンプリングを用いて word2vec を事前学習する。まずは、Section 18.3 で説明した `d2l.load_data_ptb` 関数を呼び出して、このデータセットのデータイテレータと語彙を取得しよう。

```
from d2l import torch as d2l
import math
import torch
from torch import nn

batch_size, max_window_size, num_noise_words = 512, 5, 5
data_iter, vocab = d2l.load_data_ptb(batch_size, max_window_size,
                                     num_noise_words)
```

```
from d2l import mxnet as d2l
import math
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
npx.set_np()

batch_size, max_window_size, num_noise_words = 512, 5, 5
data_iter, vocab = d2l.load_data_ptb(batch_size, max_window_size,
                                     num_noise_words)
```

```
x = d2l.tensor([[1, 2, 3], [4, 5, 6]])
embed(x)
```

```
x = d2l.tensor([[1, 2, 3], [4, 5, 6]])
embed(x)
```

18.4.1 Skip-Gram モデル

ここでは、埋め込み層とバッチ行列積を用いて skip-gram モデルを実装する。まず、埋め込み層がどのように動作するかを復習しよう。

埋め込み層

Section 13.7 で説明したように、埋め込み層はトークンのインデックスをその特徴ベクトルに対応付ける。この層の重みは、行数が辞書サイズ（`input_dim`）に等しく、列数が各トークンのベクトル次元（`output_dim`）に等しい行列である。単語埋め込みモデルを学習した後には、この重みが必要になる。

```
embed = nn.Embedding(num_embeddings=20, embedding_dim=4)
print(f'Parameter embedding_weight ({embed.weight.shape}, '
      f'dtype={embed.weight.dtype})')
```

```
Parameter embedding_weight (torch.Size([20, 4]), dtype=torch.float32)
```

```
embed = nn.Embedding(input_dim=20, output_dim=4)
embed.initialize()
embed.weight
```

```
[07:56:22] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
Parameter embedding0_weight (shape=(20, 4), dtype=float32)
```

```
pred = d2l.tensor([[1.1, -2.2, 3.3, -4.4]] * 2)
label = d2l.tensor([[1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0]])
mask = d2l.tensor([[1, 1, 1, 1], [1, 1, 0, 0]])
loss(pred, label, mask) * mask.shape[1] / mask.sum(axis=1)
```

```
pred = d2l.tensor([[1.1, -2.2, 3.3, -4.4]] * 2)
label = d2l.tensor([[1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0]])
mask = d2l.tensor([[1, 1, 1, 1], [1, 1, 0, 0]])
loss(pred, label, mask) * mask.shape[1] / mask.sum(axis=1)
```

埋め込み層の入力は、トークン（単語）のインデックスである。任意のトークンインデックス i に対して、そのベクトル表現は埋め込み層の重み行列の i^{th} 行から得られる。ベクトル次元（output_dim）を 4 に設定したので、形状が (2, 3) のトークンインデックスのミニバッチに対して、埋め込み層は形状 (2, 3, 4) のベクトルを返す。

```
x = d2l.tensor([[1, 2, 3], [4, 5, 6]])
embed(x)
```

```
tensor([[[ 0.4381,  0.3144, -0.4938,  0.2273],
         [-2.5282,  0.2348, -0.2010, -0.0579],
         [ 0.7516, -0.2634,  0.5099, -1.5015]],

        [[-0.2473,  2.6848, -0.0227,  0.9302],
         [ 0.7180, -0.2187, -0.5601, -0.7886],
         [ 0.1330, -0.8095,  0.0933,  0.5542]]], grad_fn=<EmbeddingBackward0>)
```

```
x = d2l.tensor([[1, 2, 3], [4, 5, 6]])
embed(x)
```

```
array([[[ 0.01438687,  0.05011239,  0.00628365,  0.04861524],
        [-0.01068833,  0.01729892,  0.02042518, -0.01618656],
        [-0.00873779, -0.02834515,  0.05484822, -0.06206018]],

       [[ 0.06491279, -0.03182812, -0.01631819, -0.00312688],
        [ 0.0408415 ,  0.04370362,  0.00404529, -0.0028032 ],
        [ 0.00952624, -0.01501013,  0.05958354,  0.04705103]]])
```

```
def sigmd(x):
    return -math.log(1 / (1 + math.exp(-x)))

print(f'{(sigmd(1.1) + sigmd(2.2) + sigmd(-3.3) + sigmd(4.4)) / 4:.4f}')
print(f'{(sigmd(-1.1) + sigmd(-2.2)) / 2:.4f}')
```

```
def sigmd(x):
    return -math.log(1 / (1 + math.exp(-x)))

print(f'{(sigmd(1.1) + sigmd(2.2) + sigmd(-3.3) + sigmd(4.4)) / 4:.4f}')
print(f'{(sigmd(-1.1) + sigmd(-2.2)) / 2:.4f}')
```

順伝播の定義

順伝播では、skip-gram モデルの入力は形状 (バッチサイズ, 1) の中心語インデックス `center` と、形状 (バッチサイズ, `max_len`) の連結された文脈語およびノイズ語インデックス `contexts_and_negatives` である。ここで `max_len` は Section 18.3.5 で定義されている。これら 2 つの変数はまず、埋め込み層を通してトークンインデックスからベクトルへ変換され、その後、バッチ行列積 (Section 14.3.2 で説明) によって形状 (バッチサイズ, 1, `max_len`) の出力が返される。出力の各要素は、中心語ベクトルと文脈語またはノイズ語ベクトルの内積である。

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = torch.bmm(v, u.permute(0, 2, 1))
    return pred
```

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = npx.batch_dot(v, u.swapaxes(1, 2))
    return pred
```

```
lr, num_epochs = 0.002, 5
train(net, data_iter, lr, num_epochs)
```

```
lr, num_epochs = 0.002, 5
train(net, data_iter, lr, num_epochs)
```

この skip_gram 関数の出力形状を、いくつかの例の入力で確認してみよう。

```
skip_gram(torch.ones((2, 1), dtype=torch.long),
          torch.ones((2, 4), dtype=torch.long), embed, embed).shape
```

```
torch.Size([2, 1, 4])
```

```
skip_gram(np.ones((2, 1)), np.ones((2, 4)), embed, embed).shape
```

```
(2, 1, 4)
```

18.4.2 学習

負例サンプリングを用いて skip-gram モデルを学習する前に、まず損失関数を定義しよう。

バイナリ交差エントロピー損失

Section 18.2.1 における負例サンプリングの損失関数の定義に従って、バイナリ交差エントロピー損失を用いる。

```
class SigmoidBCELoss(nn.Module):
    # Binary cross-entropy loss with masking
    def __init__(self):
        super().__init__()

    def forward(self, inputs, target, mask=None):
        out = nn.functional.binary_cross_entropy_with_logits(
            inputs, target, weight=mask, reduction="none")
        return out.mean(dim=1)

loss = SigmoidBCELoss()
```

```
loss = gluon.loss.SigmoidBCELoss()
```

Section 18.3.5 で説明した mask 変数と label 変数を思い出しよ。以下では、与えられた変数に対するバイナリ交差エントロピー損失を計算する。

```
pred = d2l.tensor([[1.1, -2.2, 3.3, -4.4]] * 2)
label = d2l.tensor([[1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0]])
mask = d2l.tensor([[1, 1, 1, 1], [1, 1, 0, 0]])
loss(pred, label, mask) * mask.shape[1] / mask.sum(axis=1)
```

```
tensor([0.9352, 1.8462])
```

以下では、上の結果が（より非効率な方法で）バイナリ交差エントロピー損失におけるシグモイド活性化関数を用いてどのように計算されるかを示す。非マスクの予測値に対して平均化された 2 つの正規化損失として考えることができる。

```
def sigmd(x):
    return -math.log(1 / (1 + math.exp(-x)))

print(f'{(sigmd(1.1) + sigmd(2.2) + sigmd(-3.3) + sigmd(4.4)) / 4:.4f}')
print(f'{(sigmd(-1.1) + sigmd(-2.2)) / 2:.4f}')
```

```
0.9352
1.8462
```

モデルパラメータの初期化

語彙中のすべての単語について、それらが中心語として使われる場合と文脈語として使われる場合に対応する 2 つの埋め込み層を定義する。単語ベクトルの次元 embed_size は 100 に設定する。

```
embed_size = 100
net = nn.Sequential(nn.Embedding(num_embeddings=len(vocab),
                                 embedding_dim=embed_size),
                    nn.Embedding(num_embeddings=len(vocab),
                                 embedding_dim=embed_size))
```

```
embed_size = 100
net = nn.Sequential()
net.add(nn.Embedding(input_dim=len(vocab), output_dim=embed_size),
        nn.Embedding(input_dim=len(vocab), output_dim=embed_size))
```

学習ループの定義

学習ループを以下に定義する。パディングが存在するため、損失関数の計算はこれまでの学習関数と少し異なる。

```
def train(net, data_iter, lr, num_epochs, device=d2l.try_gpu()):
    def init_weights(module):
        if type(module) == nn.Embedding:
            nn.init.xavier_uniform_(module.weight)
    net.apply(init_weights)
    net = net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
```

(continues on next page)

(continued from previous page)

```
                            xlim=[1, num_epochs])
    # Sum of normalized losses, no. of normalized losses
    metric = d2l.Accumulator(2)
    for epoch in range(num_epochs):
        timer, num_batches = d2l.Timer(), len(data_iter)
        for i, batch in enumerate(data_iter):
            optimizer.zero_grad()
            center, context_negative, mask, label = [
                data.to(device) for data in batch]

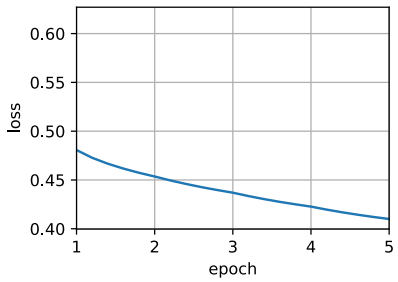
            pred = skip_gram(center, context_negative, net[0], net[1])
            l = (loss(pred.reshape(label.shape).float(), label.float(), mask)
                     / mask.sum(axis=1) * mask.shape[1])
            l.sum().backward()
            optimizer.step()
            metric.add(l.sum(), l.numel())
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, '
          f'{metric[1] / timer.stop():.1f} tokens/sec on {str(device)}')
```

```
def train(net, data_iter, lr, num_epochs, device=d2l.try_gpu()):
    net.initialize(ctx=device, force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs])
    # Sum of normalized losses, no. of normalized losses
    metric = d2l.Accumulator(2)
    for epoch in range(num_epochs):
        timer, num_batches = d2l.Timer(), len(data_iter)
        for i, batch in enumerate(data_iter):
            center, context_negative, mask, label = [
                data.as_in_ctx(device) for data in batch]
            with autograd.record():
                pred = skip_gram(center, context_negative, net[0], net[1])
                l = (loss(pred.reshape(label.shape), label, mask) *
                     mask.shape[1] / mask.sum(axis=1))
            l.backward()
            trainer.step(batch_size)
            metric.add(l.sum(), l.size)
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, '
          f'{metric[1] / timer.stop():.1f} tokens/sec on {str(device)}')
```

これで、負例サンプリングを用いた skip-gram モデルを学習できる。

```
lr, num_epochs = 0.002, 5
train(net, data_iter, lr, num_epochs)
```

```
loss 0.410, 223842.3 tokens/sec on cuda:0
```



18.4.3 単語埋め込みの適用

word2vec モデルを学習した後は、学習済みモデルの単語ベクトルのコサイン類似度を用いて、入力単語と意味的に最も近い単語を辞書から見つけることができる。

```
def get_similar_tokens(query_token, k, embed):
    W = embed.weight.data
    x = W[vocab[query_token]]
    # Compute the cosine similarity. Add 1e-9 for numerical stability
    cos = torch.mv(W, x) / torch.sqrt(torch.sum(W * W, dim=1) *
                                      torch.sum(x * x) + 1e-9)
    topk = torch.topk(cos, k=k+1)[1].cpu().numpy().astype('int32')
    for i in topk[1:]:  # Remove the input words
        print(f'cosine sim={float(cos[i]):.3f}: {vocab.to_tokens(i)}')

get_similar_tokens('chip', 3, net[0])
```

```
cosine sim=0.784: microprocessor
cosine sim=0.712: chips
cosine sim=0.708: intel
```

```
def get_similar_tokens(query_token, k, embed):
    W = embed.weight.data()
    x = W[vocab[query_token]]
    # Compute the cosine similarity. Add 1e-9 for numerical stability
    cos = np.dot(W, x) / np.sqrt(np.sum(W * W, axis=1) * np.sum(x * x) + 1e-9)
    topk = npx.topk(cos, k=k+1, ret_typ='indices').asnumpy().astype('int32')
    for i in topk[1:]:  # Remove the input words
        print(f'cosine sim={float(cos[i]):.3f}: {vocab.to_tokens(i)}')

get_similar_tokens('chip', 3, net[0])
```

```
cosine sim=0.688: microprocessor
```

(continues on next page)

(continued from previous page)

```
cosine sim=0.658: workstations
cosine sim=0.647: intel
```

18.4.4 まとめ

- 埋め込み層とバイナリ交差エントロピー損失を用いて、負例サンプリング付きの skip-gram モデルを学習できる。
- 単語埋め込みの応用として、単語ベクトルのコサイン類似度に基づいて、与えられた単語に意味的に類似した単語を見つけることができる。

18.4.5 演習

1. 学習済みモデルを用いて、他の入力単語に対して意味的に類似した単語を見つけてみよう。ハイパーパラメータを調整することで結果を改善できるか？
2. 学習コーパスが非常に大きい場合、現在のミニバッチ内の中心語に対する文脈語とノイズ語を、モデルパラメータを更新するときにサンプリングすることがよくある。言い換えると、同じ中心語でも学習エポックごとに異なる文脈語やノイズ語を持ちうるということである。この方法の利点は何だろうか？この学習方法を実装してみよう。

18.5 グローバルベクトル（GloVe）による単語埋め込み

コンテキストウィンドウ内での単語同士の共起は、豊かな意味情報を含んでいる可能性がある。たとえば、大規模コーパスでは、単語“solid”は“steam”よりも“ice”と共起しやすい一方で、単語“gas”は“ice”よりも“steam”とより頻繁に共起するだろう。さらに、このような共起の大域的なコーパス統計は事前に計算できるため、より効率的な学習につながる。単語埋め込みのためにコーパス全体の統計情報を活用するため、まず Section 18.1.3 のスキップグラムモデルを、共起回数のような大域的なコーパス統計を用いて解釈し直そう。

18.5.1 大域的コーパス統計を用いたスキップグラム

スキップグラムモデルにおいて、単語 w_i が与えられたときの単語 w_j の条件付き確率 $P(w_j \mid w_i)$ を q_{ij} と表すと、次式を得る。

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)}, \tag{18.5.1}$$

ここで、任意のインデックス i に対して、ベクトル $\mathbf{v}_i$ と $\mathbf{u}_i$ はそれぞれ中心語および文脈語として単語 w_i を表す。また、$\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$ は語彙のインデックス集合である。

コーパス中で複数回出現しうる単語 w_i を考える。コーパス全体で、w_i が中心語として取られるすべてのコンテキストウィンドウに含まれる文脈語は、単語インデックスの 多重集合 C_i を形成する。多重集合とは、同じ要素が複数回現れることを許す集合である。各要素について、その出現回数を 重複度と呼ぶ。例として、単語 w_i がコーパス中で 2 回出現し、その 2 つのコンテキストウィンドウで w_i を中心語とする文脈語のインデックスがそれぞれ k, j, m, k と k, l, k, j であったとする。このとき、多重集合 $C_i = \{j, j, k, k, k, k, l, m\}$ となり、要素 j, k, l, m の重複度はそれぞれ 2, 4, 1, 1 である。

ここで、多重集合 C_i における要素 j の重複度を x_{ij} と表す。これは、コーパス全体において、同じコンテキストウィンドウ内で単語 w_j（文脈語）と単語 w_i（中心語）が共起する回数である。このような大域的なコーパス統計を用いると、スキップグラムモデルの損失関数は次式と等価になる。

$$-\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}} x_{ij}\log q_{ij}. \tag{18.5.2}$$

さらに、x_i を、w_i が中心語として現れるコンテキストウィンドウ内のすべての文脈語の数、すなわち $|C_i|$ と等しい量として定義する。中心語 w_i が与えられたときに文脈語 w_j を生成する条件付き確率を x_{ij}/x_i として p_{ij} とおくと、(18.5.2) は次のように書き換えられる。

$$-\sum_{i\in\mathcal{V}} x_i \sum_{j\in\mathcal{V}} p_{ij}\log q_{ij}. \tag{18.5.3}$$

(18.5.3) では、$-\sum_{j\in\mathcal{V}} p_{ij}\log q_{ij}$ が、大域的コーパス統計から得られる条件付き分布 p_{ij} と、モデル予測の条件付き分布 q_{ij} のクロスエントロピーを計算している。さらにこの損失は、上で説明したように x_i によって重み付けされている。(18.5.3) の損失関数を最小化すると、予測された条件付き分布が大域的コーパス統計から得られる条件付き分布に近づく。

確率分布間の距離を測るために一般的に用いられるとはいえ、クロスエントロピー損失関数はここでは良い選択ではないかもしれない。第一に、Section 18.2 で述べたように、q_{ij} を適切に正規化するためのコストは語彙全体にわたる和を必要とし、計算量が大きくなりうる。第二に、大規模コーパスから得られる多数の稀な事象が、クロスエントロピー損失では過度に大きな重みを与えられてしまうことが多い。

18.5.2 GloVe モデル

この観点から、*GloVe* モデルは、二乗損失に基づいてスキップグラムモデルに 3 つの変更を加える (Pennington *et al.*, 2014):

1. 確率分布ではない変数 $p'_{ij} = x_{ij}$ と $q'_{ij} = \exp(\mathbf{u}_j^\top \mathbf{v}_i)$ を用い、両者の対数を取ることで、二乗損失項は $\left(\log p'_{ij} - \log q'_{ij}\right)^2 = \left(\mathbf{u}_j^\top \mathbf{v}_i - \log x_{ij}\right)^2$ となる。

2. 各単語 w_i に対して 2 つのスカラーのモデルパラメータを追加する。すなわち、中心語バイアス b_i と文脈語バイアス c_i である。

3. 各損失項の重みを重み関数 $h(x_{ij})$ に置き換える。ここで $h(x)$ は区間 $[0, 1]$ で増加関数である。

これらをまとめると、GloVe の学習は次の損失関数を最小化することに等しい。

$$\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} h(x_{ij}) \left(\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log x_{ij} \right)^2 . \tag{18.5.4}$$

重み関数としては、$x < c$（たとえば $c = 100$）のとき $h(x) = (x/c)^\alpha$（たとえば $\alpha = 0.75$）、それ以外では $h(x) = 1$ とする選択が提案されている。この場合、$h(0) = 0$ なので、$x_{ij} = 0$ に対する二乗損失項は計算効率のため省略できる。たとえば、ミニバッチ確率的勾配降下法で学習する場合、各反復で 非ゼロの x_{ij} をランダムにミニバッチとしてサンプリングし、勾配を計算してモデルパラメータを更新する。これらの非ゼロの x_{ij} は事前計算された大域的コーパス統計であるため、このモデルは *Global Vectors* の略で GloVe と呼ばれる。

強調しておくべきなのは、単語 w_i が単語 w_j のコンテキストウィンドウに現れるなら、その逆も成り立つということである。したがって、$x_{ij} = x_{ji}$ である。非対称な条件付き確率 p_{ij} に適合する word2vec とは異なり、GloVe は対称な $\log x_{ij}$ に適合する。したがって、GloVe モデルでは、任意の単語の中心語ベクトルと文脈語ベクトルは数学的に等価である。しかし実際には、初期化値が異なるため、学習後も同じ単語がこの 2 つのベクトルで異なる値を持つことがある。GloVe では、それらを出力ベクトルとして和を取る。

18.5.3 共起確率の比から GloVe を解釈する

GloVe モデルは別の観点からも解釈できる。Section 18.5.1 と同じ記法を用いて、$p_{ij} \stackrel{\text{def}}{=} P(w_j \mid w_i)$ を、コーパス中で単語 w_i を中心語として与えたときに文脈語 w_j を生成する条件付き確率とする。tab_glove には、単語“ice”と“steam”に対するいくつかの共起確率と、その比が大規模コーパスの統計に基づいて示されている。

: 大規模コーパスから得られた単語同士の共起確率とその比（Pennington *et al.* (2014) の表 1 を改変）

Table 18.5.1: label:tab_glove

w_k=	solid	gas	water	fashion
$p_1 = P(w_k \mid \text{ice})$	0.00019	0.000066	0.003	0.000017
$p_2 = P(w_k \mid \text{steam})$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

tab_glove から次のことが観察できる。

- “ice”に関連し“steam”には関連しない単語 w_k、たとえば w_k = solid については、8.9 のような大きな共起確率比が期待される。
- “steam”に関連し“ice”には関連しない単語 w_k、たとえば w_k = gas については、0.085 のような小さな共起確率比が期待される。
- “ice”と“steam”の両方に関連する単語 w_k、たとえば w_k = water については、1.36 のように 1 に近い共起確率比が期待される。
- “ice”と“steam”のどちらにも関連しない単語 w_k、たとえば w_k = fashion については、0.96 のように 1 に近い共起確率比が期待される。

共起確率の比は、単語間の関係を直感的に表現できることがわかる。したがって、この比に適合するように、3 つの単語ベクトルの関数を設計できる。中心語が w_i、文脈語が w_j と w_k であるときの共起確率比 p_{ij}/p_{ik} に対して、何らかの関数 f を用いてこの比に適合させたい。

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}. \tag{18.5.5}$$

f の設計には多くの可能性があるが、以下では妥当な選択だけを取り上げる。共起確率の比はスカラーであるため、f もスカラー関数である必要がある。たとえば $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f\left((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i\right)$ のようにである。(18.5.5) で単語インデックス j と k を入れ替えると、$f(x)f(-x) = 1$ が成り立たなければならない。したがって、1 つの可能性は $f(x) = \exp(x)$ であり、すなわち

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right)}{\exp\left(\mathbf{u}_k^\top \mathbf{v}_i\right)} \approx \frac{p_{ij}}{p_{ik}}. \tag{18.5.6}$$

ここで、$\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right) \approx \alpha p_{ij}$ とおく。ここで α は定数である。$p_{ij} = x_{ij}/x_i$ なので、両辺の対数を取ると $\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$ を得る。$-\log \alpha + \log x_i$ を適合させるために、中心語バイアス b_i や文脈語バイアス c_j のような追加のバイアス項を用いることができる。

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log x_{ij}. \tag{18.5.7}$$

(18.5.7) の二乗誤差を重み付きで測ると、(18.5.4) の GloVe 損失関数が得られる。

18.5.4 まとめ

- スキップグラムモデルは、単語同士の共起回数のような大域的コーパス統計を用いて解釈できる。
- クロスエントロピー損失は、特に大規模コーパスでは、2 つの確率分布の差を測るのに適した選択ではないかもしれない。GloVe は、事前計算された大域的コーパス統計に適合するように二乗損失を用いる。
- GloVe では、任意の単語について中心語ベクトルと文脈語ベクトルは数学的に等価である。

- GloVe は、単語同士の共起確率の比から解釈できる。

18.5.5 演習

1. 単語 w_i と w_j が同じコンテキストウィンドウ内で共起する場合、テキスト系列におけるそれらの距離をどのように用いて、条件付き確率 p_{ij} を計算する方法を再設計できるか。ヒント: GloVe 論文 (Pennington *et al.*, 2014) の第 4.2 節を参照せよ。
2. 任意の単語について、その中心語バイアスと文脈語バイアスは GloVe において数学的に等価か。なぜか？

18.6 サブワード埋め込み

英語では、“helps”、“helped”、“helping”のような単語は、同じ単語“help”の屈折形である。“dog”と“dogs”の関係は“cat”と“cats”の関係と同じであり、“boy”と“boyfriend”の関係は“girl”と“girlfriend”の関係と同じである。フランス語やスペイン語のような他の言語では、多くの動詞が 40 を超える屈折形を持ち、フィンランド語では、名詞が最大 15 の格を持つことがある。言語学では、形態論は語形成と言語間の関係を研究する。しかし、単語の内部構造は word2vec においても GloVe においても調べられていなかった。

18.6.1 fastText モデル

word2vec における単語表現を思い出そう。skip-gram モデルでも continuous bag-of-words モデルでも、同じ単語の異なる屈折形は共有パラメータを持たない別々のベクトルとして直接表現される。形態情報を利用するために、*fastText* モデルは サブワード埋め込みのアプローチを提案した。ここでサブワードとは文字 n-gram のことである (Bojanowski *et al.*, 2017)。単語レベルのベクトル表現を学習する代わりに、fastText はサブワードレベルの skip-gram とみなすことができる。各 中心語は、そのサブワードベクトルの和で表現される。

fastText で各中心語のサブワードをどのように得るかを、単語“where”を使って説明しよう。まず、接頭辞と接尾辞を他のサブワードと区別するために、単語の先頭と末尾に特別な文字“<”と“>”を追加する。次に、単語から文字 n-gram を抽出する。たとえば、$n = 3$ のとき、長さ 3 のすべてのサブワード“<wh”、“whe”、“her”、“ere”、“re>”、および特別なサブワード“<where>”を得る。

fastText では、任意の単語 w に対して、$\mathcal{G}_w$ を長さ 3 から 6 のすべてのサブワードと特別なサブワードの和集合とする。語彙はすべての単語のサブワードの和集合である。辞書中のサブワード g のベクトルを $\mathbf{z}_g$ とすると、skip-gram モデルにおける中心語としての単語 w のベクトル $\mathbf{v}_w$ はそのサブワードベクトルの和で

ある：

$$\mathbf{v}_w = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g. \tag{18.6.1}$$

fastText の残りの部分は skip-gram モデルと同じである。skip-gram モデルと比べると、fastText の語彙はより大きく、その結果、モデルパラメータも増える。さらに、単語の表現を計算するには、そのすべてのサブワードベクトルを加算しなければならず、計算量が高くなる。しかし、似た構造を持つ単語間でサブワードのパラメータが共有されるおかげで、頻度の低い単語や、さらには語彙外単語であっても、fastText ではより良いベクトル表現を得られる可能性がある。

18.6.2 バイトペアエンコーディング

fastText では、抽出されるサブワードはすべて 3 から 6 のような指定された長さでなければならないため、語彙サイズを事前に定義できない。固定サイズの語彙で可変長サブワードを扱うために、*byte pair encoding* (BPE) と呼ばれる圧縮アルゴリズムを用いてサブワードを抽出できる (Sennrich *et al.*, 2015)。

バイトペアエンコーディングは、訓練データセットに対して統計的分析を行い、単語内の共通シンボル、たとえば任意の長さの連続する文字列を見つけ出す。長さ 1 のシンボルから始めて、バイトペアエンコーディングは最も頻出する連続シンボルのペアを反復的に結合し、より長い新しいシンボルを生成する。効率のため、単語境界をまたぐペアは考慮しないことに注意しよう。最終的に、このようなシンボルをサブワードとして単語を分割できる。バイトペアエンコーディングとその変種は、GPT-2 (Radford *et al.*, 2019) や RoBERTa (Liu *et al.*, 2019) のような人気の自然言語処理の事前学習モデルにおける入力表現として用いられている。以下では、バイトペアエンコーディングの仕組みを説明する。

まず、シンボルの語彙を、英語の小文字すべて、特別な語末シンボル `'_'`、および特別な未知語シンボル `'[UNK]'` として初期化する。

```
import collections

symbols = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
           'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
           '_', '[UNK]']
```

単語の境界をまたぐシンボルのペアは考慮しないので、データセット内での単語とその頻度（出現回数）を対応づける辞書 `raw_token_freqs` だけが必要である。特別なシンボル `'_'` は各単語の末尾に付加されるため、出力シンボル列（たとえば“a_ tall er_ man”）から単語列（たとえば“a taller man”）を容易に復元できる。結合処理を単一文字と特別シンボルだけからなる語彙で始めるので、各単語内の隣接する文字の間（辞書 `token_freqs` のキー）には空白を挿入する。言い換えると、空白は単語内のシンボルの区切り文字である。

```
raw_token_freqs = {'fast_': 4, 'faster_': 3, 'tall_': 5, 'taller_': 4}
token_freqs = {}
for token, freq in raw_token_freqs.items():
    token_freqs[' '.join(list(token))] = raw_token_freqs[token]
token_freqs
```

```
{'f a s t _': 4, 'f a s t e r _': 3, 't a l l _': 5, 't a l l e r _': 4}
```

以下の get_max_freq_pair 関数を定義する。これは、入力辞書 token_freqs のキーに含まれる単語から、単語内で最も頻出する連続シンボルのペアを返す。

```
def get_max_freq_pair(token_freqs):
    pairs = collections.defaultdict(int)
    for token, freq in token_freqs.items():
        symbols = token.split()
        for i in range(len(symbols) - 1):
            # Key of `pairs` is a tuple of two consecutive symbols
            pairs[symbols[i], symbols[i + 1]] += freq
    return max(pairs, key=pairs.get)  # Key of `pairs` with the max value
```

連続シンボルの頻度に基づく貪欲法として、バイトペアエンコーディングは以下の merge_symbols 関数を用いて、最も頻出する連続シンボルのペアを結合し、新しいシンボルを生成する。

```
def merge_symbols(max_freq_pair, token_freqs, symbols):
    symbols.append(''.join(max_freq_pair))
    new_token_freqs = dict()
    for token, freq in token_freqs.items():
        new_token = token.replace(' '.join(max_freq_pair),
                                  ''.join(max_freq_pair))
        new_token_freqs[new_token] = token_freqs[token]
    return new_token_freqs
```

ここで、辞書 token_freqs のキーに対してバイトペアエンコーディングアルゴリズムを反復的に適用しよう。最初の反復では、最も頻出する連続シンボルのペアは 't' と 'a' なので、バイトペアエンコーディングはそれらを結合して新しいシンボル 'ta' を生成する。2 回目の反復では、バイトペアエンコーディングは 'ta' と 'l' の結合を続け、別の新しいシンボル 'tal' を得る。

```
num_merges = 10
for i in range(num_merges):
    max_freq_pair = get_max_freq_pair(token_freqs)
    token_freqs = merge_symbols(max_freq_pair, token_freqs, symbols)
    print(f'merge #{i + 1}:', max_freq_pair)
```

```
merge #1: ('t', 'a')
```

(continues on next page)

(continued from previous page)

```
merge #2: ('ta', 'l')
merge #3: ('tal', 'l')
merge #4: ('f', 'a')
merge #5: ('fa', 's')
merge #6: ('fas', 't')
merge #7: ('e', 'r')
merge #8: ('er', '_')
merge #9: ('tall', '_')
merge #10: ('fast', '_')
```

バイトペアエンコーディングを 10 回反復した後、リスト symbols には、他のシンボルから反復的に結合された 10 個のシンボルが追加されていることがわかる。

```
print(symbols)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p
↪', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '_', '[UNK]', 'ta', 'tal
↪', 'tall', 'fa', 'fas', 'fast', 'er', 'er_', 'tall_', 'fast_']
```

辞書 raw_token_freqs のキーで指定された同じデータセットに対して、バイトペアエンコーディングアルゴリズムの結果として、各単語はサブワード“fast_”、“fast”、“er_”、“tall_”、および“tall”に分割される。たとえば、単語“faster_”と“taller_”はそれぞれ“fast er_”と“tall er_”に分割される。

```
print(list(token_freqs.keys()))
```

```
['fast_', 'fast er_', 'tall_', 'tall er_']
```

バイトペアエンコーディングの結果は、使用するデータセットに依存することに注意しよう。あるデータセットで学習したサブワードを使って、別のデータセットの単語を分割することもできる。貪欲法として、以下の segment_BPE 関数は、入力引数 symbols から可能な限り長いサブワードに単語を分割しようとする。

```
def segment_BPE(tokens, symbols):
    outputs = []
    for token in tokens:
        start, end = 0, len(token)
        cur_output = []
        # Segment token with the longest possible subwords from symbols
        while start < len(token) and start < end:
            if token[start: end] in symbols:
                cur_output.append(token[start: end])
                start = end
                end = len(token)
```

(continues on next page)

(continued from previous page)

```
            else:
                end -= 1
        if start < len(token):
            cur_output.append('[UNK]')
        outputs.append(' '.join(cur_output))
    return outputs
```

以下では、前述のデータセットから学習したリスト `symbols` 内のサブワードを用いて、別のデータセットを表す `tokens` を分割する。

```
tokens = ['tallest_', 'fatter_']
print(segment_BPE(tokens, symbols))
```

```
['tall e s t _', 'fa t t er_']
```

18.6.3 まとめ

- fastText モデルはサブワード埋め込みのアプローチを提案する。word2vec の skip-gram モデルに基づき、中心語をそのサブワードベクトルの和として表現する。
- バイトペアエンコーディングは、訓練データセットを統計的に分析して、単語内の共通シンボルを見つけ出す。貪欲法として、バイトペアエンコーディングは最も頻出する連続シンボルのペアを反復的に結合する。
- サブワード埋め込みは、頻度の低い単語や辞書外単語の表現品質を改善する可能性がある。

18.6.4 演習

1. 例として、英語には約 3×10^8 個の可能な 6-gram がある。サブワードが多すぎるとどのような問題が生じるか。どう対処すればよいか。ヒント: fastText 論文 (Bojanowski *et al.*, 2017) の Section 3.2 の末尾を参照せよ。
2. continuous bag-of-words モデルに基づくサブワード埋め込みモデルをどのように設計するか。
3. サイズ m の語彙を得るには、初期シンボル語彙サイズが n のとき、何回の結合操作が必要か。
4. バイトペアエンコーディングの考え方を拡張してフレーズを抽出するにはどうすればよいか。

18.7 単語の類似性とアナロジー

Section 18.4 では、
小さなデータセットで word2vec モデルを学習し、
入力単語に対して意味的に類似した単語を見つけるために適用した。
実際には、大規模コーパスで事前学習された単語ベクトルは、
後続の自然言語処理タスクに適用できる。
これについては、後ほど Section 19 で扱いる。
大規模コーパスから得られた事前学習済み単語ベクトルの意味を
わかりやすく示すために、
単語の類似性とアナロジーのタスクに適用してみよう。

```
from d2l import torch as d2l
import torch
from torch import nn
import os
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
import os

npx.set_np()
```

```
#@save
d2l.DATA_HUB['glove.6b.50d'] = (d2l.DATA_URL + 'glove.6B.50d.zip',
                                '0b8703943ccdb6eb788e6f091b8946e82231bc4d')

#@save
d2l.DATA_HUB['glove.6b.100d'] = (d2l.DATA_URL + 'glove.6B.100d.zip',
                                 'cd43bfb07e44e6f27cbcc7bc9ae3d80284fdaf5a')

#@save
d2l.DATA_HUB['glove.42b.300d'] = (d2l.DATA_URL + 'glove.42B.300d.zip',
                                  'b5116e234e9eb9076672cfeabf5469f3eec904fa')

#@save
d2l.DATA_HUB['wiki.en'] = (d2l.DATA_URL + 'wiki.en.zip',
                           'c1816da3821ae9f43899be655002f6c723e91b88')
```

```
#@save
d2l.DATA_HUB['glove.6b.50d'] = (d2l.DATA_URL + 'glove.6B.50d.zip',
                                '0b8703943ccdb6eb788e6f091b8946e82231bc4d')

#@save
```

(continues on next page)

(continued from previous page)

```
d2l.DATA_HUB['glove.6b.100d'] = (d2l.DATA_URL + 'glove.6B.100d.zip',
                                 'cd43bfb07e44e6f27cbcc7bc9ae3d80284fdaf5a')

#@save
d2l.DATA_HUB['glove.42b.300d'] = (d2l.DATA_URL + 'glove.42B.300d.zip',
                                  'b5116e234e9eb9076672cfeabf5469f3eec904fa')

#@save
d2l.DATA_HUB['wiki.en'] = (d2l.DATA_URL + 'wiki.en.zip',
                           'c1816da3821ae9f43899be655002f6c723e91b88')
```

18.7.1 事前学習済み単語ベクトルの読み込み

以下に、次元 50、100、300 の事前学習済み GloVe 埋め込みを示す。
これらは GloVe website[114] からダウンロードできる。
事前学習済み fastText 埋め込みは複数の言語で利用できる。
ここでは英語版の 1 つ（300 次元の“wiki.en”）を扱い、
fastText website[115] からダウンロードできる。

114 

115 

```
#@save
d2l.DATA_HUB['glove.6b.50d'] = (d2l.DATA_URL + 'glove.6B.50d.zip',
                                '0b8703943ccdb6eb788e6f091b8946e82231bc4d')

#@save
d2l.DATA_HUB['glove.6b.100d'] = (d2l.DATA_URL + 'glove.6B.100d.zip',
                                 'cd43bfb07e44e6f27cbcc7bc9ae3d80284fdaf5a')

#@save
d2l.DATA_HUB['glove.42b.300d'] = (d2l.DATA_URL + 'glove.42B.300d.zip',
                                  'b5116e234e9eb9076672cfeabf5469f3eec904fa')

#@save
d2l.DATA_HUB['wiki.en'] = (d2l.DATA_URL + 'wiki.en.zip',
                           'c1816da3821ae9f43899be655002f6c723e91b88')
```

```
#@save
d2l.DATA_HUB['glove.6b.50d'] = (d2l.DATA_URL + 'glove.6B.50d.zip',
                                '0b8703943ccdb6eb788e6f091b8946e82231bc4d')

#@save
d2l.DATA_HUB['glove.6b.100d'] = (d2l.DATA_URL + 'glove.6B.100d.zip',
                                 'cd43bfb07e44e6f27cbcc7bc9ae3d80284fdaf5a')

#@save
d2l.DATA_HUB['glove.42b.300d'] = (d2l.DATA_URL + 'glove.42B.300d.zip',
                                  'b5116e234e9eb9076672cfeabf5469f3eec904fa')

#@save
```

(continues on next page)

(continued from previous page)

```
d2l.DATA_HUB['wiki.en'] = (d2l.DATA_URL + 'wiki.en.zip',
                           'c1816da3821ae9f43899be655002f6c723e91b88')
```

```
#@save
class TokenEmbedding:
    """Token Embedding."""
    def __init__(self, embedding_name):
        self.idx_to_token, self.idx_to_vec = self._load_embedding(
            embedding_name)
        self.unknown_idx = 0
        self.token_to_idx = {token: idx for idx, token in
                             enumerate(self.idx_to_token)}

    def _load_embedding(self, embedding_name):
        idx_to_token, idx_to_vec = ['<unk>'], []
        data_dir = d2l.download_extract(embedding_name)
        # GloVe website: https://nlp.stanford.edu/projects/glove/
        # fastText website: https://fasttext.cc/
        with open(os.path.join(data_dir, 'vec.txt'), 'r') as f:
            for line in f:
                elems = line.rstrip().split(' ')
                token, elems = elems[0], [float(elem) for elem in elems[1:]]
                # Skip header information, such as the top row in fastText
                if len(elems) > 1:
                    idx_to_token.append(token)
                    idx_to_vec.append(elems)
        idx_to_vec = [[0] * len(idx_to_vec[0])] + idx_to_vec
        return idx_to_token, d2l.tensor(idx_to_vec)

    def __getitem__(self, tokens):
        indices = [self.token_to_idx.get(token, self.unknown_idx)
                   for token in tokens]
        vecs = self.idx_to_vec[d2l.tensor(indices)]
        return vecs

    def __len__(self):
        return len(self.idx_to_token)
```

```
#@save
class TokenEmbedding:
    """Token Embedding."""
    def __init__(self, embedding_name):
        self.idx_to_token, self.idx_to_vec = self._load_embedding(
            embedding_name)
        self.unknown_idx = 0
        self.token_to_idx = {token: idx for idx, token in
                             enumerate(self.idx_to_token)}

    def _load_embedding(self, embedding_name):
        idx_to_token, idx_to_vec = ['<unk>'], []
        data_dir = d2l.download_extract(embedding_name)
        # GloVe website: https://nlp.stanford.edu/projects/glove/
        # fastText website: https://fasttext.cc/
```

(continues on next page)

(continued from previous page)

```
        with open(os.path.join(data_dir, 'vec.txt'), 'r') as f:
            for line in f:
                elems = line.rstrip().split(' ')
                token, elems = elems[0], [float(elem) for elem in elems[1:]]
                # Skip header information, such as the top row in fastText
                if len(elems) > 1:
                    idx_to_token.append(token)
                    idx_to_vec.append(elems)
        idx_to_vec = [[0] * len(idx_to_vec[0])] + idx_to_vec
        return idx_to_token, d2l.tensor(idx_to_vec)

    def __getitem__(self, tokens):
        indices = [self.token_to_idx.get(token, self.unknown_idx)
                   for token in tokens]
        vecs = self.idx_to_vec[d2l.tensor(indices)]
        return vecs

    def __len__(self):
        return len(self.idx_to_token)
```

これらの事前学習済み GloVe および fastText 埋め込みを読み込むために、
以下の `TokenEmbedding` クラスを定義する。

```
#@save
class TokenEmbedding:
    """Token Embedding."""
    def __init__(self, embedding_name):
        self.idx_to_token, self.idx_to_vec = self._load_embedding(
            embedding_name)
        self.unknown_idx = 0
        self.token_to_idx = {token: idx for idx, token in
                             enumerate(self.idx_to_token)}

    def _load_embedding(self, embedding_name):
        idx_to_token, idx_to_vec = ['<unk>'], []
        data_dir = d2l.download_extract(embedding_name)
        # GloVe website: https://nlp.stanford.edu/projects/glove/
        # fastText website: https://fasttext.cc/
        with open(os.path.join(data_dir, 'vec.txt'), 'r') as f:
            for line in f:
                elems = line.rstrip().split(' ')
                token, elems = elems[0], [float(elem) for elem in elems[1:]]
                # Skip header information, such as the top row in fastText
                if len(elems) > 1:
                    idx_to_token.append(token)
                    idx_to_vec.append(elems)
        idx_to_vec = [[0] * len(idx_to_vec[0])] + idx_to_vec
        return idx_to_token, d2l.tensor(idx_to_vec)

    def __getitem__(self, tokens):
```

(continues on next page)

(continued from previous page)

```
        indices = [self.token_to_idx.get(token, self.unknown_idx)
                   for token in tokens]
        vecs = self.idx_to_vec[d2l.tensor(indices)]
        return vecs

    def __len__(self):
        return len(self.idx_to_token)
```

```
#@save
class TokenEmbedding:
    """Token Embedding."""
    def __init__(self, embedding_name):
        self.idx_to_token, self.idx_to_vec = self._load_embedding(
            embedding_name)
        self.unknown_idx = 0
        self.token_to_idx = {token: idx for idx, token in
                             enumerate(self.idx_to_token)}

    def _load_embedding(self, embedding_name):
        idx_to_token, idx_to_vec = ['<unk>'], []
        data_dir = d2l.download_extract(embedding_name)
        # GloVe website: https://nlp.stanford.edu/projects/glove/
        # fastText website: https://fasttext.cc/
        with open(os.path.join(data_dir, 'vec.txt'), 'r') as f:
            for line in f:
                elems = line.rstrip().split(' ')
                token, elems = elems[0], [float(elem) for elem in elems[1:]]
                # Skip header information, such as the top row in fastText
                if len(elems) > 1:
                    idx_to_token.append(token)
                    idx_to_vec.append(elems)
        idx_to_vec = [[0] * len(idx_to_vec[0])] + idx_to_vec
        return idx_to_token, d2l.tensor(idx_to_vec)

    def __getitem__(self, tokens):
        indices = [self.token_to_idx.get(token, self.unknown_idx)
                   for token in tokens]
        vecs = self.idx_to_vec[d2l.tensor(indices)]
        return vecs

    def __len__(self):
        return len(self.idx_to_token)
```

```
glove_6b50d = TokenEmbedding('glove.6b.50d')
```

```
glove_6b50d = TokenEmbedding('glove.6b.50d')
```

以下では、
50 次元の GloVe 埋め込み（Wikipedia の一部で事前学習済み）を読み込みる。

TokenEmbedding インスタンスを作成するとき、
指定した埋め込みファイルがまだ存在しなければダウンロードされる。

```
glove_6b50d = TokenEmbedding('glove.6b.50d')
```

```
Downloading ../data/glove.6B.50d.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/glove.6B.50d.zip...
```

```
glove_6b50d = TokenEmbedding('glove.6b.50d')
```

```
Downloading ../data/glove.6B.50d.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/glove.6B.50d.zip...
[07:14:52] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
len(glove_6b50d)
```

```
len(glove_6b50d)
```

語彙サイズを出力する。語彙には 400000 語（トークン）と特別な未知語トークンが含まれる。

```
len(glove_6b50d)
```

```
400001
```

```
len(glove_6b50d)
```

```
400001
```

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

語彙中の単語のインデックスを取得したり、その逆を行ったりできる。

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

```
(3367, 'beautiful')
```

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

```
(3367, 'beautiful')
```

```
def get_similar_tokens(query_token, k, embed):
    topk, cos = knn(embed.idx_to_vec, embed[[query_token]], k + 1)
    for i, c in zip(topk[1:], cos[1:]):  # Exclude the input word
        print(f'cosine sim={float(c):.3f}: {embed.idx_to_token[int(i)]}')
```


```
def get_similar_tokens(query_token, k, embed):
    topk, cos = knn(embed.idx_to_vec, embed[[query_token]], k + 1)
    for i, c in zip(topk[1:], cos[1:]):  # Exclude the input word
        print(f'cosine sim={float(c):.3f}: {embed.idx_to_token[int(i)]}')
```


18.7.2 事前学習済み単語ベクトルの適用

読み込んだ GloVe ベクトルを用いて、
以下の単語類似性およびアナロジーのタスクに適用することで、
その意味を示す。

単語の類似性

Section 18.4.3 と同様に、
単語ベクトル間のコサイン類似度に基づいて
入力単語に意味的に類似した単語を見つけるために、
以下の knn
（k-最近傍）関数を実装する。

```
def knn(W, x, k):
    # Add 1e-9 for numerical stability
    cos = torch.mv(W, x.reshape(-1,)) / (
        torch.sqrt(torch.sum(W * W, axis=1) + 1e-9) *
        torch.sqrt((x * x).sum()))
    _, topk = torch.topk(cos, k=k)
    return topk, [cos[int(i)] for i in topk]
```

```
def knn(W, x, k):
    # Add 1e-9 for numerical stability
    cos = np.dot(W, x.reshape(-1,)) / (
```

(continues on next page)

(continued from previous page)

```
        np.sqrt(np.sum(W * W, axis=1) + 1e-9) * np.sqrt((x * x).sum()))
    topk = npx.topk(cos, k=k, ret_typ='indices')
    return topk, [cos[int(i)] for i in topk]
```

```
get_similar_tokens('chip', 3, glove_6b50d)
```

```
get_similar_tokens('chip', 3, glove_6b50d)
```

次に、TokenEmbedding インスタンス embed に含まれる事前学習済み単語ベクトルを使って、
類似単語を検索する。

```
def get_similar_tokens(query_token, k, embed):
    topk, cos = knn(embed.idx_to_vec, embed[[query_token]], k + 1)
    for i, c in zip(topk[1:], cos[1:]):  # Exclude the input word
        print(f'cosine sim={float(c):.3f}: {embed.idx_to_token[int(i)]}')
```

```
def get_similar_tokens(query_token, k, embed):
    topk, cos = knn(embed.idx_to_vec, embed[[query_token]], k + 1)
    for i, c in zip(topk[1:], cos[1:]):  # Exclude the input word
        print(f'cosine sim={float(c):.3f}: {embed.idx_to_token[int(i)]}')
```

```
get_similar_tokens('baby', 3, glove_6b50d)
```

```
get_similar_tokens('baby', 3, glove_6b50d)
```

事前学習済み単語ベクトル glove_6b50d の語彙には 400000 語と特別な未知語トークンが含まれる。
入力単語と未知語トークンを除いて、
この語彙の中から
単語“chip”に意味的に最も類似した 3 語を見つけてみよう。

```
get_similar_tokens('chip', 3, glove_6b50d)
```

```
cosine sim=0.856: chips
cosine sim=0.749: intel
cosine sim=0.749: electronics
```

```
get_similar_tokens('chip', 3, glove_6b50d)
```

```
cosine sim=0.856: chips
cosine sim=0.749: intel
cosine sim=0.749: electronics
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

以下は“baby”と“beautiful”に類似した単語を出力する。

```
get_similar_tokens('baby', 3, glove_6b50d)
```

```
cosine sim=0.839: babies
cosine sim=0.800: boy
cosine sim=0.792: girl
```

```
get_similar_tokens('baby', 3, glove_6b50d)
```

```
cosine sim=0.839: babies
cosine sim=0.800: boy
cosine sim=0.792: girl
```

```
def get_analogy(token_a, token_b, token_c, embed):
    vecs = embed[[token_a, token_b, token_c]]
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = knn(embed.idx_to_vec, x, 1)
    return embed.idx_to_token[int(topk[0])]  # Remove unknown words
```

```
def get_analogy(token_a, token_b, token_c, embed):
    vecs = embed[[token_a, token_b, token_c]]
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = knn(embed.idx_to_vec, x, 1)
    return embed.idx_to_token[int(topk[0])]  # Remove unknown words
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

```
cosine sim=0.921: lovely
cosine sim=0.893: gorgeous
cosine sim=0.830: wonderful
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

```
cosine sim=0.921: lovely
cosine sim=0.893: gorgeous
cosine sim=0.830: wonderful
```

```
get_analogy('man', 'woman', 'son', glove_6b50d)
```

```
get_analogy('man', 'woman', 'son', glove_6b50d)
```

単語のアナロジー

類似単語を見つけるだけでなく、
単語ベクトルを単語アナロジーのタスクにも適用できる。
たとえば、
“man”:“woman”::“son”:“daughter”
は単語アナロジーの形式である。
つまり、
“man”は“woman”に対して、
“son”は“daughter”に対する、という関係である。
具体的には、
単語アナロジー補完タスクは次のように定義できる。
単語アナロジー $a : b :: c : d$ に対して、最初の 3 語 a, b, c が与えられたとき、d を見つける。
単語 w のベクトルを $\text{vec}(w)$ と表す。
このアナロジーを完成させるために、
$\text{vec}(c) + \text{vec}(b) - \text{vec}(a)$ の結果に最も類似したベクトルを持つ単語を見つける。

```
def get_analogy(token_a, token_b, token_c, embed):
    vecs = embed[[token_a, token_b, token_c]]
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = knn(embed.idx_to_vec, x, 1)
    return embed.idx_to_token[int(topk[0])]  # Remove unknown words
```

```
def get_analogy(token_a, token_b, token_c, embed):
    vecs = embed[[token_a, token_b, token_c]]
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = knn(embed.idx_to_vec, x, 1)
    return embed.idx_to_token[int(topk[0])]  # Remove unknown words
```

```
get_analogy('beijing', 'china', 'tokyo', glove_6b50d)
```

```
get_analogy('beijing', 'china', 'tokyo', glove_6b50d)
```

読み込んだ単語ベクトルを使って、“male-female”のアナロジーを確認してみよう。

```
get_analogy('man', 'woman', 'son', glove_6b50d)
```

```
'daughter'
```

```
get_analogy('man', 'woman', 'son', glove_6b50d)
```

```
'daughter'
```

```
get_analogy('bad', 'worst', 'big', glove_6b50d)
```

```
get_analogy('bad', 'worst', 'big', glove_6b50d)
```

以下は
“capital-country”のアナロジーを完成させる。
“beijing”:“china”::“tokyo”:“japan”。
これは、
事前学習済み単語ベクトルに意味情報が含まれていることを示している。

```
get_analogy('beijing', 'china', 'tokyo', glove_6b50d)
```

```
'japan'
```

```
get_analogy('beijing', 'china', 'tokyo', glove_6b50d)
```

```
'japan'
```

```
get_analogy('do', 'did', 'go', glove_6b50d)
```

```
get_analogy('do', 'did', 'go', glove_6b50d)
```

“bad”:“worst”::“big”:“biggest”
のような
“形容詞-最上級形容詞”のアナロジーでは、
事前学習済み単語ベクトルが
統語情報を捉えている可能性があることがわかる。

```
get_analogy('bad', 'worst', 'big', glove_6b50d)
```

```
'biggest'
```

事前学習済み単語ベクトルにおける過去形の概念が捉えられていることを示すために、
“do”:“did”::“go”:“went”
という
“現在形-過去形”のアナロジーで構文をテストできる。

```
get_analogy('do', 'did', 'go', glove_6b50d)
```

```
'went'
```

18.7.3 まとめ

- 実際には、大規模コーパスで事前学習された単語ベクトルは、後続の自然言語処理タスクに適用できる。
- 事前学習済み単語ベクトルは、単語の類似性とアナロジーのタスクに適用できる。

18.7.4 演習

1. `TokenEmbedding('wiki.en')` を使って fastText の結果を試しよ。
2. 語彙が非常に大きい場合、類似単語の検索や単語アナロジーの補完をより高速に行うにはどうすればよいだろうか？

18.8 Transformer による双方向エンコーダ表現（BERT）

これまでに、自然言語理解のためのいくつかの単語埋め込みモデルを紹介してきた。事前学習後、その出力は行ごとに所定の語彙の単語を表すベクトルが並んだ行列とみなせる。実際、これらの単語埋め込みモデルはいずれも 文脈非依存である。まず、この性質を示そう。

18.8.1 文脈非依存から文脈依存へ

Section 18.4 と Section 18.7 の実験を思い出そう。たとえば、word2vec と GloVe は、単語の文脈に関係なく（文脈がある場合でも）同じ単語に同じ事前学習済みベクトルを割り当てる。形式的には、任意のトークン x の文脈非依存表現とは、入力として x のみを受け取る関数 $f(x)$ である。自然言語には多義性や複雑な意味論が豊富に存在するため、文脈非依存表現には明らかな限界がある。たとえば、“a crane is flying”と“a crane driver came”という文脈における“crane”は、まったく異なる意味を持つ。したがって、同じ単語でも文脈に応じて異なる表現が割り当てられることがある。

このことが、単語の表現がその文脈に依存する 文脈依存単語表現の発展を促した。したがって、トークン x の文脈依存表現とは、x とその文脈 $c(x)$ の両方に依存する関数 $f(x, c(x))$ である。代表的な文脈依存表現には、TagLM（language-model-augmented sequence tagger）(Peters *et al.*, 2017)、CoVe（Context Vectors）(McCann *et al.*, 2017)、ELMo（Embeddings from Language Models）(Peters *et al.*, 2018) がある。

たとえば、系列全体を入力として与えることで、ELMo は入力系列中の各単語に表現を割り当てる関数である。具体的には、ELMo は事前学習済みの双方向 LSTM のすべての中間層表現を出力表現として組み合わせる。その後、ELMo 表現は、既存の教師ありモデルに追加特徴量として加えられる。たとえば、ELMo 表現と既存モデルにおけるトークンの元の表現（例：GloVe）を連結する。一方で、ELMo 表現を追加した後は、事前学習済み双方向 LSTM モデルのすべての重みは固定される。他方で、既存の教師ありモデルは、特定のタスクに合わせて個別に調整される。当時のタスクごとの最良モデルを活用することで、ELMo の追加により、感情分析、自然言語推論、意味役割付与、共参照解決、固有表現認識、質問応答の 6 つの自然言語処理タスクで最先端性能が向上した。

18.8.2 タスク特化からタスク非依存へ

ELMo は多様な自然言語処理タスクの解決を大きく改善したが、それでも各解法は タスク特化のアーキテクチャに依存している。しかし、自然言語処理タスクごとに個別のアーキテクチャを設計するのは、実際には容易ではない。GPT（Generative Pre-Training）モデルは、文脈依存表現のための一般的な タスク非依存モデルを設計しようとする試みである (Radford *et al.*, 2018)。Transformer デコーダ上に構築された GPT は、テキスト系列の表現に用いられる言語モデルを事前学習する。GPT を下流タスクに適用する際には、言語モデルの出力が追加の線形出力層に入力され、タスクのラベルを予測する。事前学習済みモデルのパラメータを固定する ELMo とは対照的に、GPT は下流タスクの教師あり学習中に、事前学習済み Transformer デコーダの すべてのパラメータを微調整する。GPT は、自然言語推論、質問応答、文類似度、分類の 12 個のタスクで評価され、モデルアーキテクチャへの変更を最小限に抑えつつ、そのうち 9 個で最先端性能を更新した。

しかし、言語モデルの自己回帰的な性質のため、GPT は前方（左から右）しか見ない。“i went to the bank to deposit cash”と“i went to the bank to sit down”という文脈では、“bank”は左側の文脈に敏感であるが、GPT は“bank”に対して同じ表現を返してしまう。そのため、意味は異なるにもかかわらず同一の表現になってしまう。

18.8.3 BERT: 両者の長所を組み合わせる

これまで見てきたように、ELMo は文脈を双方向に符号化するが、タスク特化のアーキテクチャを用いる。一方、GPT はタスク非依存だが、文脈を左から右へ符号化する。両者の長所を組み合わせた BERT（Bidirectional Encoder Representations from Transformers）は、文脈を双方向に符号化し、幅広い自然言語処理タスクに対して最小限のアーキテクチャ変更しか必要としない (Devlin *et al.*, 2018)。事前学習済み Transformer エンコーダを用いることで、BERT は双方向の文脈に基づいて任意のトークンを表現できる。下流タスクの教師あり学習では、BERT は 2 つの点で GPT に似ている。第 1 に、BERT の表現は追加の出力層に入力され、タスクの性質に応じて、たとえば各トークンを予測するか系列全体を予測するかといった最小限のアーキテクチャ変更で済む。第 2 に、事前学習済み Transformer エンコーダのすべてのパラメータが微調整され、追加の出力層はゼロから学習される。Fig. 18.8.1 は ELMo、GPT、BERT の違いを示している。

BERT はさらに、(i) 単一テキスト分類（例：感情分析）、(ii) テキスト対分類（例：自然言語推論）、(iii) 質問応答、(iv) テキストタグ付け（例：固有表現認識）という広いカテゴリにわたる 11 個の自然言語処理タスクで最先端性能を更新した。2018 年に提案された、文脈依存の ELMo からタスク非依存の GPT と BERT まで、自然言語の深い表現を事前学習するという概念的には単純だが実証的には強力な手法は、さまざまな自然言語処理タスクの解決を一変させた。

この章の残りでは、BERT の事前学習を詳しく見ていく。自然言語処理アプリケ

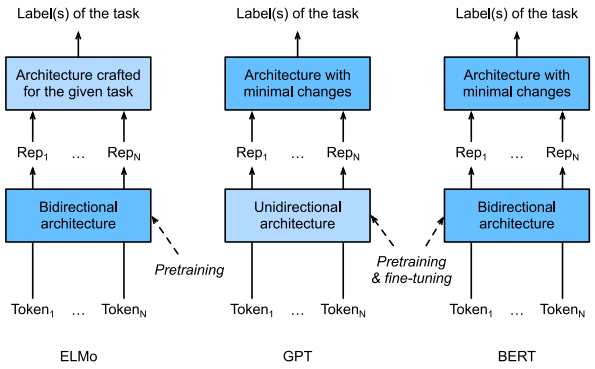


Fig. 18.8.1 ELMo、GPT、BERT の比較。

ーションについては Section 19 で説明する際に、下流アプリケーション向けの BERT の微調整を示す。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn

npx.set_np()
```

```
#@save
def get_tokens_and_segments(tokens_a, tokens_b=None):
    """Get tokens of the BERT input sequence and their segment IDs."""
    tokens = ['<cls>'] + tokens_a + ['<sep>']
    # 0 and 1 are marking segment A and B, respectively
    segments = [0] * (len(tokens_a) + 2)
    if tokens_b is not None:
        tokens += tokens_b + ['<sep>']
        segments += [1] * (len(tokens_b) + 1)
    return tokens, segments
```

```
#@save
def get_tokens_and_segments(tokens_a, tokens_b=None):
    """Get tokens of the BERT input sequence and their segment IDs."""
    tokens = ['<cls>'] + tokens_a + ['<sep>']
    # 0 and 1 are marking segment A and B, respectively
    segments = [0] * (len(tokens_a) + 2)
    if tokens_b is not None:
        tokens += tokens_b + ['<sep>']
        segments += [1] * (len(tokens_b) + 1)
    return tokens, segments
```

18.8.4 入力表現

自然言語処理では、感情分析のように単一テキストを入力とするタスクもあれば、自然言語推論のように 2 つのテキスト系列の対を入力とするタスクもある。BERT の入力系列は、単一テキストとテキスト対の両方を明確に表現できる。前者では、BERT の入力系列は特別な分類トークン“<cls>”、テキスト系列のトークン、および特別な区切りトークン“<sep>”の連結である。後者では、BERT の入力系列は“<cls>”、第 1 テキスト系列のトークン、“<sep>”、第 2 テキスト系列のトークン、そして“<sep>”の連結である。ここでは、“BERT input sequence”という用語を他の種類の“sequences”と一貫して区別する。たとえば、1 つの *BERT input sequence* は、1 つの *text sequence* または 2 つの *text sequences* のいずれかを含みうる。

テキスト対を区別するために、学習されるセグメント埋め込み $\mathbf{e}_A$ と $\mathbf{e}_B$ が、それぞれ第 1 系列と第 2 系列のトークン埋め込みに加えられる。単一テキスト入力では、$\mathbf{e}_A$ のみが使われる。

以下の `get_tokens_and_segments` は、1 文または 2 文を入力として受け取り、BERT 入力系列のトークンと対応するセグメント ID を返す。

```
#@save
def get_tokens_and_segments(tokens_a, tokens_b=None):
    """Get tokens of the BERT input sequence and their segment IDs."""
    tokens = ['<cls>'] + tokens_a + ['<sep>']
    # 0 and 1 are marking segment A and B, respectively
    segments = [0] * (len(tokens_a) + 2)
    if tokens_b is not None:
        tokens += tokens_b + ['<sep>']
        segments += [1] * (len(tokens_b) + 1)
    return tokens, segments
```

BERT は双方向アーキテクチャとして Transformer エンコーダを採用する。Transformer エンコーダで一般的なように、位置埋め込みは BERT 入力系列の各位置に加えられる。ただし、元の Transformer エンコーダとは異なり、BERT では 学習可能な位置埋め込みを用いる。要するに、Fig. 18.8.2 が示すように、BERT 入力系列の埋め込みはトークン埋め込み、セグメント埋め込み、位置埋め込みの和である。

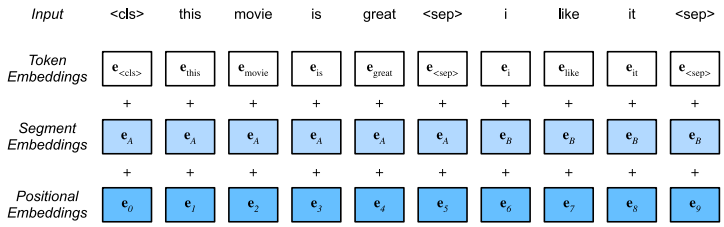


Fig. 18.8.2 BERT 入力系列の埋め込みは、トークン埋め込み、セグメント埋め込み、位置埋め込みの和である。

以下の BERTEncoder クラスは、Section 14.7 で実装した TransformerEncoder クラスに似ている。TransformerEncoder とは異なり、BERTEncoder はセグメント埋め込みと学習可能な位置埋め込みを用いる。

```
#@save
class BERTEncoder(nn.Module):
    """BERT encoder."""
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                 num_blks, dropout, max_len=1000, **kwargs):
        super(BERTEncoder, self).__init__(**kwargs)
        self.token_embedding = nn.Embedding(vocab_size, num_hiddens)
        self.segment_embedding = nn.Embedding(2, num_hiddens)
        self.blks = nn.Sequential()
        for i in range(num_blks):
            self.blks.add_module(f"{i}", d2l.TransformerEncoderBlock(
                num_hiddens, ffn_num_hiddens, num_heads, dropout, True))
        # In BERT, positional embeddings are learnable, thus we create a
        # parameter of positional embeddings that are long enough
        self.pos_embedding = nn.Parameter(torch.randn(1, max_len,
                                                      num_hiddens))

    def forward(self, tokens, segments, valid_lens):
        # Shape of `X` remains unchanged in the following code snippet:
        # (batch size, max sequence length, `num_hiddens`)
        X = self.token_embedding(tokens) + self.segment_embedding(segments)
        X = X + self.pos_embedding[:, :X.shape[1], :]
        for blk in self.blks:
            X = blk(X, valid_lens)
        return X
```

```
#@save
class BERTEncoder(nn.Block):
    """BERT encoder."""
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                 num_blks, dropout, max_len=1000, **kwargs):
        super(BERTEncoder, self).__init__(**kwargs)
        self.token_embedding = nn.Embedding(vocab_size, num_hiddens)
        self.segment_embedding = nn.Embedding(2, num_hiddens)
        self.blks = nn.Sequential()
        for _ in range(num_blks):
            self.blks.add(d2l.TransformerEncoderBlock(
                num_hiddens, ffn_num_hiddens, num_heads, dropout, True))
        # In BERT, positional embeddings are learnable, thus we create a
        # parameter of positional embeddings that are long enough
        self.pos_embedding = self.params.get('pos_embedding',
                                             shape=(1, max_len, num_hiddens))

    def forward(self, tokens, segments, valid_lens):
        # Shape of `X` remains unchanged in the following code snippet:
        # (batch size, max sequence length, `num_hiddens`)
        X = self.token_embedding(tokens) + self.segment_embedding(segments)
        X = X + self.pos_embedding.data(ctx=X.ctx)[:, :X.shape[1], :]
        for blk in self.blks:
            X = blk(X, valid_lens)
        return X
```

語彙サイズが 10000 だと仮定しよう。BERTEncoder の順伝播による推論を示すために、そのインスタンスを作成し、パラメータを初期化する。

```
vocab_size, num_hiddens, ffn_num_hiddens, num_heads = 10000, 768, 1024, 4
ffn_num_input, num_blks, dropout = 768, 2, 0.2
encoder = BERTEncoder(vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                      num_blks, dropout)
```

```
vocab_size, num_hiddens, ffn_num_hiddens, num_heads = 10000, 768, 1024, 4
num_blks, dropout = 2, 0.2
encoder = BERTEncoder(vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                      num_blks, dropout)
encoder.initialize()
```

```
[07:06:23] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

tokens を長さ 8 の 2 つの BERT 入力系列と定義し、各トークンは語彙のインデックスであるとする。入力 tokens に対する BERTEncoder の順伝播推論は、各トークンがベクトルで表現された符号化結果を返す。そのベクトル長はハイパーパラメータ num_hiddens によって定義される。このハイパーパラメータは通常、Transformer エンコーダの 隠れサイズ（隠れユニット数）と呼ばれる。

```
tokens = torch.randint(0, vocab_size, (2, 8))
segments = torch.tensor([[0, 0, 0, 0, 1, 1, 1, 1], [0, 0, 0, 1, 1, 1, 1, 1]])
encoded_X = encoder(tokens, segments, None)
encoded_X.shape
```

```
torch.Size([2, 8, 768])
```

```
tokens = np.random.randint(0, vocab_size, (2, 8))
segments = np.array([[0, 0, 0, 0, 1, 1, 1, 1], [0, 0, 0, 1, 1, 1, 1, 1]])
encoded_X = encoder(tokens, segments, None)
encoded_X.shape
```

```
(2, 8, 768)
```

18.8.5 事前学習タスク

BERTEncoder の順伝播推論は、入力テキストの各トークンと挿入された特別トークン“<cls>”および“<seq>”の BERT 表現を与える。次に、これらの表現を用いて BERT の事前学習の損失関数を計算する。事前学習は次の 2 つのタスクから構成される：マスク付き言語モデルと次文予測である。

マスク付き言語モデル

Section 12.3 で示したように、言語モデルは左側の文脈を用いてトークンを予測する。各トークンの表現のために双方向の文脈を符号化するには、BERT はトークンをランダムにマスクし、双方向文脈からのトークンを用いて自己教師ありでマスクされたトークンを予測する。このタスクは マスク付き言語モデルと呼ばれる。

この事前学習タスクでは、トークンの 15% がランダムに選ばれ、予測対象のマスクされたトークンとなる。ラベルを使って答えを覗き見しないように、マスクされたトークンを予測するための単純な方法は、BERT 入力系列中でそれを常に特別な“<mask>”トークンに置き換えることである。しかし、この人工的な特別トークン“<mask>”は微調整では決して現れない。事前学習と微調整の間のこのような不一致を避けるため、あるトークンが予測のためにマスクされる場合（たとえば“this movie is great”で“great”がマスクおよび予測対象として選ばれた場合）、入力では次のように置き換えられる。

- 80% の確率で特別な“<mask>”トークンに置き換える（例：“this movie is great”が“this movie is <mask>”になる）；
- 10% の確率でランダムなトークンに置き換える（例：“this movie is great”が“this movie is drink”になる）；
- 10% の確率でラベルのトークンをそのまま残す（例：“this movie is great”が“this movie is great”のままになる）。

15% のうち 10% の確率でランダムなトークンが挿入されることに注意しよう。このときどき入るノイズは、BERT が双方向文脈の符号化においてマスクされたトークンに過度に偏らないよう促す（特にラベルのトークンがそのまま残る場合）。

以下の `MaskLM` クラスを実装して、BERT 事前学習のマスク付き言語モデルタスクにおけるマスクされたトークンを予測する。予測には 1 隠れ層 MLP（`self.mlp`）を用いる。順伝播推論では、2 つの入力を受け取る：`BERTEncoder` の符号化結果と、予測対象のトークン位置である。出力は、これらの位置における予測結果である。

```
#@save
class MaskLM(nn.Module):
    """The masked language model task of BERT."""
    def __init__(self, vocab_size, num_hiddens, **kwargs):
        super(MaskLM, self).__init__(**kwargs)
        self.mlp = nn.Sequential(nn.LazyLinear(num_hiddens),
                                 nn.ReLU(),
                                 nn.LayerNorm(num_hiddens),
                                 nn.LazyLinear(vocab_size))

    def forward(self, X, pred_positions):
        num_pred_positions = pred_positions.shape[1]
        pred_positions = pred_positions.reshape(-1)
```

(continues on next page)

(continued from previous page)

```
        batch_size = X.shape[0]
        batch_idx = torch.arange(0, batch_size)
        # Suppose that `batch_size` = 2, `num_pred_positions` = 3, then
        # `batch_idx` is `torch.tensor([0, 0, 0, 1, 1, 1])`
        batch_idx = torch.repeat_interleave(batch_idx, num_pred_positions)
        masked_X = X[batch_idx, pred_positions]
        masked_X = masked_X.reshape((batch_size, num_pred_positions, -1))
        mlm_Y_hat = self.mlp(masked_X)
        return mlm_Y_hat
```

```
#@save
class MaskLM(nn.Block):
    """The masked language model task of BERT."""
    def __init__(self, vocab_size, num_hiddens, **kwargs):
        super(MaskLM, self).__init__(**kwargs)
        self.mlp = nn.Sequential()
        self.mlp.add(
            nn.Dense(num_hiddens, flatten=False, activation='relu'))
        self.mlp.add(nn.LayerNorm())
        self.mlp.add(nn.Dense(vocab_size, flatten=False))

    def forward(self, X, pred_positions):
        num_pred_positions = pred_positions.shape[1]
        pred_positions = pred_positions.reshape(-1)
        batch_size = X.shape[0]
        batch_idx = np.arange(0, batch_size)
        # Suppose that `batch_size` = 2, `num_pred_positions` = 3, then
        # `batch_idx` is `np.array([0, 0, 0, 1, 1, 1])`
        batch_idx = np.repeat(batch_idx, num_pred_positions)
        masked_X = X[batch_idx, pred_positions]
        masked_X = masked_X.reshape((batch_size, num_pred_positions, -1))
        mlm_Y_hat = self.mlp(masked_X)
        return mlm_Y_hat
```

MaskLM の順伝播推論を示すために、そのインスタンス mlm を作成して初期化する。BERTEncoder の順伝播推論から得られる encoded_X は、2 つの BERT 入力系列を表していることを思い出そう。mlm_positions を、encoded_X のいずれかの BERT 入力系列において予測する 3 つのインデックスとして定義する。mlm の順伝播推論は、encoded_X のすべてのマスク位置 mlm_positions における予測結果 mlm_Y_hat を返す。各予測について、結果のサイズは語彙サイズに等しい。

```
mlm = MaskLM(vocab_size, num_hiddens)
mlm_positions = torch.tensor([[1, 5, 2], [6, 1, 5]])
mlm_Y_hat = mlm(encoded_X, mlm_positions)
mlm_Y_hat.shape
```

```
torch.Size([2, 3, 10000])
```

```
mlm = MaskLM(vocab_size, num_hiddens)
mlm.initialize()
mlm_positions = np.array([[1, 5, 2], [6, 1, 5]])
mlm_Y_hat = mlm(encoded_X, mlm_positions)
mlm_Y_hat.shape
```

```
(2, 3, 10000)
```

マスク下で予測されたトークン `mlm_Y_hat` の正解ラベル `mlm_Y` があれば、BERT事前学習におけるマスク付き言語モデルタスクの交差エントロピー損失を計算できる。

```
mlm_Y = torch.tensor([[7, 8, 9], [10, 20, 30]])
loss = nn.CrossEntropyLoss(reduction='none')
mlm_l = loss(mlm_Y_hat.reshape((-1, vocab_size)), mlm_Y.reshape(-1))
mlm_l.shape
```

```
torch.Size([6])
```

```
mlm_Y = np.array([[7, 8, 9], [10, 20, 30]])
loss = gluon.loss.SoftmaxCrossEntropyLoss()
mlm_l = loss(mlm_Y_hat.reshape((-1, vocab_size)), mlm_Y.reshape(-1))
mlm_l.shape
```

```
(6,)
```

次文予測

マスク付き言語モデルは単語の表現のために双方向文脈を符号化できるが、テキスト対の論理的関係を明示的にはモデル化しない。2 つのテキスト系列の関係を理解する助けとして、BERT は事前学習において 次文予測という二値分類タスクを考える。事前学習用の文対を生成する際、半分の確率では実際に連続する文であり、ラベルは“True”である。残りの半分では、第 2 文はコーパスからランダムにサンプリングされ、ラベルは“False”である。

以下の `NextSentencePred` クラスは、1 隠れ層 MLP を用いて、BERT 入力系列における第 2 文が第 1 文の次の文であるかどうかを予測する。Transformer エンコーダの自己注意により、特別トークン“<cls>”の BERT 表現は入力中の 2 つの文の両方を符号化する。したがって、MLP 分類器の出力層（`self.output`）は `X` を入力として受け取る。ここで `X` は、符号化された“<cls>”トークンを入力とする MLP の隠れ層の出力である。

```
#@save
class NextSentencePred(nn.Module):
```

(continues on next page)

(continued from previous page)

```
    """The next sentence prediction task of BERT."""
    def __init__(self, **kwargs):
        super(NextSentencePred, self).__init__(**kwargs)
        self.output = nn.LazyLinear(2)

    def forward(self, X):
        # `X` shape: (batch size, `num_hiddens`)
        return self.output(X)
```

```
#@save
class NextSentencePred(nn.Block):
    """The next sentence prediction task of BERT."""
    def __init__(self, **kwargs):
        super(NextSentencePred, self).__init__(**kwargs)
        self.output = nn.Dense(2)

    def forward(self, X):
        # `X` shape: (batch size, `num_hiddens`)
        return self.output(X)
```

NextSentencePred のインスタンスの順伝播推論は、各 BERT 入力系列に対して二値予測を返すことがわかる。

```
# PyTorch by default will not flatten the tensor as seen in mxnet where, if
# flatten=True, all but the first axis of input data are collapsed together
encoded_X = torch.flatten(encoded_X, start_dim=1)
# input_shape for NSP: (batch size, `num_hiddens`)
nsp = NextSentencePred()
nsp_Y_hat = nsp(encoded_X)
nsp_Y_hat.shape
```

```
torch.Size([2, 2])
```

```
nsp = NextSentencePred()
nsp.initialize()
nsp_Y_hat = nsp(encoded_X)
nsp_Y_hat.shape
```

```
(2, 2)
```

2 つの二値分類の交差エントロピー損失も計算できる。

```
nsp_y = torch.tensor([0, 1])
nsp_l = loss(nsp_Y_hat, nsp_y)
nsp_l.shape
```

```
torch.Size([2])
```

```
nsp_y = np.array([0, 1])
nsp_l = loss(nsp_Y_hat, nsp_y)
nsp_l.shape
```

```
(2,)
```

前述の 2 つの事前学習タスクにおけるすべてのラベルは、手作業のラベル付けを行わずとも、事前学習コーパスから自明に得られることに注目すべきである。元の BERT は BookCorpus (Zhu *et al.*, 2015) と英語版 Wikipedia を連結したものを用いて事前学習されている。これら 2 つのテキストコーパスは非常に大きく、それぞれ 8 億語と 25 億語を含む。

18.8.6 まとめてみよう

BERT を事前学習するとき、最終的な損失関数はマスク付き言語モデルと次文予測の両方の損失関数の線形結合である。ここで、`BERTEncoder`、`MaskLM`、`NextSentencePred` の 3 つのクラスをインスタンス化することで、`BERTModel` クラスを定義できる。順伝播推論は、符号化された BERT 表現 `encoded_X`、マスク付き言語モデルの予測 `mlm_Y_hat`、および次文予測 `nsp_Y_hat` を返す。

```
#@save
class BERTModel(nn.Module):
    """The BERT model."""
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens,
                 num_heads, num_blks, dropout, max_len=1000):
        super(BERTModel, self).__init__()
        self.encoder = BERTEncoder(vocab_size, num_hiddens, ffn_num_hiddens,
                                   num_heads, num_blks, dropout,
                                   max_len=max_len)
        self.hidden = nn.Sequential(nn.LazyLinear(num_hiddens),
                                    nn.Tanh())
        self.mlm = MaskLM(vocab_size, num_hiddens)
        self.nsp = NextSentencePred()

    def forward(self, tokens, segments, valid_lens=None, pred_positions=None):
        encoded_X = self.encoder(tokens, segments, valid_lens)
        if pred_positions is not None:
            mlm_Y_hat = self.mlm(encoded_X, pred_positions)
        else:
            mlm_Y_hat = None
        # The hidden layer of the MLP classifier for next sentence prediction.
        # 0 is the index of the '<cls>' token
        nsp_Y_hat = self.nsp(self.hidden(encoded_X[:, 0, :]))
        return encoded_X, mlm_Y_hat, nsp_Y_hat
```

```
#@save
class BERTModel(nn.Block):
    """The BERT model."""
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                 num_blks, dropout, max_len=1000):
        super(BERTModel, self).__init__()
        self.encoder = BERTEncoder(vocab_size, num_hiddens, ffn_num_hiddens,
                                   num_heads, num_blks, dropout, max_len)
        self.hidden = nn.Dense(num_hiddens, activation='tanh')
        self.mlm = MaskLM(vocab_size, num_hiddens)
        self.nsp = NextSentencePred()

    def forward(self, tokens, segments, valid_lens=None, pred_positions=None):
        encoded_X = self.encoder(tokens, segments, valid_lens)
        if pred_positions is not None:
            mlm_Y_hat = self.mlm(encoded_X, pred_positions)
        else:
            mlm_Y_hat = None
        # The hidden layer of the MLP classifier for next sentence prediction.
        # 0 is the index of the '<cls>' token
        nsp_Y_hat = self.nsp(self.hidden(encoded_X[:, 0, :]))
        return encoded_X, mlm_Y_hat, nsp_Y_hat
```

18.8.7 まとめ

- word2vec や GloVe のような単語埋め込みモデルは文脈非依存である。これらは、単語の文脈に関係なく（文脈がある場合でも）同じ単語に同じ事前学習済みベクトルを割り当てる。多義性や自然言語の複雑な意味論をうまく扱うのは難しい。
- ELMo や GPT のような文脈依存単語表現では、単語の表現はその文脈に依存する。
- ELMo は文脈を双方向に符号化するが、タスク特化のアーキテクチャを用いる（ただし、自然言語処理タスクごとに個別のアーキテクチャを作るのは実際には容易ではない）。一方、GPT はタスク非依存だが、文脈を左から右へ符号化する。
- BERT は両者の長所を組み合わせる。すなわち、文脈を双方向に符号化し、幅広い自然言語処理タスクに対して最小限のアーキテクチャ変更しか必要としない。
- BERT 入力系列の埋め込みは、トークン埋め込み、セグメント埋め込み、位置埋め込みの和である。
- BERT の事前学習は 2 つのタスクから成る：マスク付き言語モデルと次文予測である。前者は単語の表現のために双方向文脈を符号化でき、後者はテキスト対の論理的関係を明示的にモデル化する。

18.8.8 演習

1. 他の条件がすべて同じなら、マスク付き言語モデルは左から右への言語モデルよりも、収束に必要な事前学習ステップ数が多くなるだろうか、それとも少なくなるだろうか。なぜか。
2. BERT の元の実装では、`BERTEncoder`（`d2l.TransformerEncoderBlock` 経由）の位置ごとのフィードフォワードネットワークと `MaskLM` の全結合層の両方で、活性化関数としてガウス誤差線形ユニット（GELU）(Hendrycks and Gimpel, 2016) が使われている。GELU と ReLU の違いについて調べよ。

18.9 BERT の事前学習のためのデータセット

Section 18.8 で実装した BERT モデルを事前学習するには、2 つの事前学習タスク、すなわちマスク付き言語モデルと次文予測を容易にするための、理想的な形式のデータセットを生成する必要がある。一方で、元の BERT モデルは BookCorpus と English Wikipedia という 2 つの巨大なコーパスの連結上で事前学習されており (Section 18.8.5 を参照)、本書の多くの読者にとっては実行が難しいである。他方で、既製の事前学習済み BERT モデルは、医療のような特定分野のアプリケーションには適さない場合がある。そのため、カスタマイズしたデータセット上で BERT を事前学習することが一般的になりつつある。BERT の事前学習を示しやすくするために、ここではより小さなコーパスである WikiText-2 (Merity *et al.*, 2016) を用いる。

Section 18.3 で word2vec の事前学習に用いた PTB データセットと比べると、WikiText-2 は (i) 元の句読点を保持しているため、次文予測に適している、(ii) 元の大文字小文字と数値を保持している、(iii) 2 倍以上大きい、という特徴がある。

```
from d2l import torch as d2l
import os
import random
import torch
```

```
from d2l import mxnet as d2l
from mxnet import gluon, np, npx
import os
import random

npx.set_np()
```

```
#@save
d2l.DATA_HUB['wikitext-2'] = (
    'https://s3.amazonaws.com/research.metamind.io/wikitext/'
    'wikitext-2-v1.zip', '3c914d17d80b1459be871a5039ac23e752a53cbe')
```

(continues on next page)

(continued from previous page)

```
#@save
def _read_wiki(data_dir):
    file_name = os.path.join(data_dir, 'wiki.train.tokens')
    with open(file_name, 'r') as f:
        lines = f.readlines()
    # Uppercase letters are converted to lowercase ones
    paragraphs = [line.strip().lower().split(' . ')
                  for line in lines if len(line.split(' . ')) >= 2]
    random.shuffle(paragraphs)
    return paragraphs
```

```
#@save
d2l.DATA_HUB['wikitext-2'] = (
    'https://s3.amazonaws.com/research.metamind.io/wikitext/'
    'wikitext-2-v1.zip', '3c914d17d80b1459be871a5039ac23e752a53cbe')

#@save
def _read_wiki(data_dir):
    file_name = os.path.join(data_dir, 'wiki.train.tokens')
    with open(file_name, 'r') as f:
        lines = f.readlines()
    # Uppercase letters are converted to lowercase ones
    paragraphs = [line.strip().lower().split(' . ')
                  for line in lines if len(line.split(' . ')) >= 2]
    random.shuffle(paragraphs)
    return paragraphs
```

WikiText-2 データセットでは、各行が 1 つの段落を表し、任意の句読点とその直前のトークンの間には空白が挿入されている。少なくとも 2 文を含む段落のみを残す。文の分割には、簡単のためにピリオドのみを区切り文字として使う。より複雑な文分割手法については、この節の最後の演習で扱いる。

```
#@save
d2l.DATA_HUB['wikitext-2'] = (
    'https://s3.amazonaws.com/research.metamind.io/wikitext/'
    'wikitext-2-v1.zip', '3c914d17d80b1459be871a5039ac23e752a53cbe')

#@save
def _read_wiki(data_dir):
    file_name = os.path.join(data_dir, 'wiki.train.tokens')
    with open(file_name, 'r') as f:
        lines = f.readlines()
    # Uppercase letters are converted to lowercase ones
    paragraphs = [line.strip().lower().split(' . ')
                  for line in lines if len(line.split(' . ')) >= 2]
    random.shuffle(paragraphs)
    return paragraphs
```

```
#@save
d2l.DATA_HUB['wikitext-2'] = (
```

(continues on next page)

(continued from previous page)

```
    'https://s3.amazonaws.com/research.metamind.io/wikitext/'
    'wikitext-2-v1.zip', '3c914d17d80b1459be871a5039ac23e752a53cbe')

#@save
def _read_wiki(data_dir):
    file_name = os.path.join(data_dir, 'wiki.train.tokens')
    with open(file_name, 'r') as f:
        lines = f.readlines()
    # Uppercase letters are converted to lowercase ones
    paragraphs = [line.strip().lower().split(' . ')
                  for line in lines if len(line.split(' . ')) >= 2]
    random.shuffle(paragraphs)
    return paragraphs
```

```
#@save
def _get_next_sentence(sentence, next_sentence, paragraphs):
    if random.random() < 0.5:
        is_next = True
    else:
        # `paragraphs` is a list of lists of lists
        next_sentence = random.choice(random.choice(paragraphs))
        is_next = False
    return sentence, next_sentence, is_next
```


```
#@save
def _get_next_sentence(sentence, next_sentence, paragraphs):
    if random.random() < 0.5:
        is_next = True
    else:
        # `paragraphs` is a list of lists of lists
        next_sentence = random.choice(random.choice(paragraphs))
        is_next = False
    return sentence, next_sentence, is_next
```


18.9.1 事前学習タスクのための補助関数の定義

以下ではまず、BERT の 2 つの事前学習タスク、次文予測とマスク付き言語モデルのための補助関数を実装する。これらの補助関数は後で、生のテキストコーパスを BERT の事前学習に適した理想的な形式のデータセットへ変換する際に呼び出される。

次文予測タスクの生成

Section 18.8.5 の説明に従い、`_get_next_sentence` 関数は二値分類タスクのための学習例を生成する。

```
#@save
def _get_next_sentence(sentence, next_sentence, paragraphs):
```

(continues on next page)

(continued from previous page)

```
    if random.random() < 0.5:
        is_next = True
    else:
        # `paragraphs` is a list of lists of lists
        next_sentence = random.choice(random.choice(paragraphs))
        is_next = False
    return sentence, next_sentence, is_next
```

```
#@save
def _get_next_sentence(sentence, next_sentence, paragraphs):
    if random.random() < 0.5:
        is_next = True
    else:
        # `paragraphs` is a list of lists of lists
        next_sentence = random.choice(random.choice(paragraphs))
        is_next = False
    return sentence, next_sentence, is_next
```

```
#@save
def _get_nsp_data_from_paragraph(paragraph, paragraphs, vocab, max_len):
    nsp_data_from_paragraph = []
    for i in range(len(paragraph) - 1):
        tokens_a, tokens_b, is_next = _get_next_sentence(
            paragraph[i], paragraph[i + 1], paragraphs)
        # Consider 1 '<cls>' token and 2 '<sep>' tokens
        if len(tokens_a) + len(tokens_b) + 3 > max_len:
            continue
        tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
        nsp_data_from_paragraph.append((tokens, segments, is_next))
    return nsp_data_from_paragraph
```

```
#@save
def _get_nsp_data_from_paragraph(paragraph, paragraphs, vocab, max_len):
    nsp_data_from_paragraph = []
    for i in range(len(paragraph) - 1):
        tokens_a, tokens_b, is_next = _get_next_sentence(
            paragraph[i], paragraph[i + 1], paragraphs)
        # Consider 1 '<cls>' token and 2 '<sep>' tokens
        if len(tokens_a) + len(tokens_b) + 3 > max_len:
            continue
        tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
        nsp_data_from_paragraph.append((tokens, segments, is_next))
    return nsp_data_from_paragraph
```

次の関数は、_get_next_sentence 関数を呼び出すことで、入力 paragraph から次文予測の学習例を生成する。ここで paragraph は文のリストであり、各文はトークンのリストである。引数 max_len は、事前学習中の BERT 入力系列の最大長を指定する。

```
#@save
def _get_nsp_data_from_paragraph(paragraph, paragraphs, vocab, max_len):
    nsp_data_from_paragraph = []
    for i in range(len(paragraph) - 1):
        tokens_a, tokens_b, is_next = _get_next_sentence(
            paragraph[i], paragraph[i + 1], paragraphs)
        # Consider 1 '<cls>' token and 2 '<sep>' tokens
        if len(tokens_a) + len(tokens_b) + 3 > max_len:
            continue
        tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
        nsp_data_from_paragraph.append((tokens, segments, is_next))
    return nsp_data_from_paragraph
```


```
#@save
def _get_nsp_data_from_paragraph(paragraph, paragraphs, vocab, max_len):
    nsp_data_from_paragraph = []
    for i in range(len(paragraph) - 1):
        tokens_a, tokens_b, is_next = _get_next_sentence(
            paragraph[i], paragraph[i + 1], paragraphs)
        # Consider 1 '<cls>' token and 2 '<sep>' tokens
        if len(tokens_a) + len(tokens_b) + 3 > max_len:
            continue
        tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
        nsp_data_from_paragraph.append((tokens, segments, is_next))
    return nsp_data_from_paragraph
```


```
#@save
def _replace_mlm_tokens(tokens, candidate_pred_positions, num_mlm_preds,
                        vocab):
    # For the input of a masked language model, make a new copy of tokens and
    # replace some of them by '<mask>' or random tokens
    mlm_input_tokens = [token for token in tokens]
    pred_positions_and_labels = []
    # Shuffle for getting 15% random tokens for prediction in the masked
    # language modeling task
    random.shuffle(candidate_pred_positions)
    for mlm_pred_position in candidate_pred_positions:
        if len(pred_positions_and_labels) >= num_mlm_preds:
            break
        masked_token = None
        # 80% of the time: replace the word with the '<mask>' token
        if random.random() < 0.8:
            masked_token = '<mask>'
        else:
            # 10% of the time: keep the word unchanged
            if random.random() < 0.5:
                masked_token = tokens[mlm_pred_position]
            # 10% of the time: replace the word with a random word
            else:
                masked_token = random.choice(vocab.idx_to_token)
        mlm_input_tokens[mlm_pred_position] = masked_token
        pred_positions_and_labels.append(
            (mlm_pred_position, tokens[mlm_pred_position]))
    return mlm_input_tokens, pred_positions_and_labels
```

```
#@save
def _replace_mlm_tokens(tokens, candidate_pred_positions, num_mlm_preds,
                        vocab):
    # For the input of a masked language model, make a new copy of tokens and
    # replace some of them by '<mask>' or random tokens
    mlm_input_tokens = [token for token in tokens]
    pred_positions_and_labels = []
    # Shuffle for getting 15% random tokens for prediction in the masked
    # language modeling task
    random.shuffle(candidate_pred_positions)
    for mlm_pred_position in candidate_pred_positions:
        if len(pred_positions_and_labels) >= num_mlm_preds:
            break
        masked_token = None
        # 80% of the time: replace the word with the '<mask>' token
        if random.random() < 0.8:
            masked_token = '<mask>'
        else:
            # 10% of the time: keep the word unchanged
            if random.random() < 0.5:
                masked_token = tokens[mlm_pred_position]
            # 10% of the time: replace the word with a random word
            else:
                masked_token = random.choice(vocab.idx_to_token)
        mlm_input_tokens[mlm_pred_position] = masked_token
        pred_positions_and_labels.append(
            (mlm_pred_position, tokens[mlm_pred_position]))
    return mlm_input_tokens, pred_positions_and_labels
```

マスク付き言語モデルタスクの生成

BERT 入力系列からマスク付き言語モデルタスクの学習例を生成するために、次の `_replace_mlm_tokens` 関数を定義する。入力において、`tokens` は BERT 入力系列を表すトークンのリスト、`candidate_pred_positions` は特殊トークンを除いた BERT 入力系列中のトークンインデックスのリストです（特殊トークンはマスク付き言語モデルタスクでは予測されない）、`num_mlm_preds` は予測数を示す（予測対象としてランダムな 15% のトークンを思い出してほしい）。Section 18.8.5 で定義したマスク付き言語モデルタスクに従い、各予測位置では、入力は特殊な“<mask>”トークンやランダムなトークンに置き換えられるか、あるいはそのまま保持される。最終的にこの関数は、置換後の入力トークン、予測が行われるトークンインデックス、およびそれらのラベルを返す。

```
#@save
def _replace_mlm_tokens(tokens, candidate_pred_positions, num_mlm_preds,
                        vocab):
    # For the input of a masked language model, make a new copy of tokens and
    # replace some of them by '<mask>' or random tokens
    mlm_input_tokens = [token for token in tokens]
    pred_positions_and_labels = []
    # Shuffle for getting 15% random tokens for prediction in the masked
    # language modeling task
```

(continues on next page)

(continued from previous page)

```
    random.shuffle(candidate_pred_positions)
    for mlm_pred_position in candidate_pred_positions:
        if len(pred_positions_and_labels) >= num_mlm_preds:
            break
        masked_token = None
        # 80% of the time: replace the word with the '<mask>' token
        if random.random() < 0.8:
            masked_token = '<mask>'
        else:
            # 10% of the time: keep the word unchanged
            if random.random() < 0.5:
                masked_token = tokens[mlm_pred_position]
            # 10% of the time: replace the word with a random word
            else:
                masked_token = random.choice(vocab.idx_to_token)
        mlm_input_tokens[mlm_pred_position] = masked_token
        pred_positions_and_labels.append(
            (mlm_pred_position, tokens[mlm_pred_position]))
    return mlm_input_tokens, pred_positions_and_labels
```

```
#@save
def _replace_mlm_tokens(tokens, candidate_pred_positions, num_mlm_preds,
                        vocab):
    # For the input of a masked language model, make a new copy of tokens and
    # replace some of them by '<mask>' or random tokens
    mlm_input_tokens = [token for token in tokens]
    pred_positions_and_labels = []
    # Shuffle for getting 15% random tokens for prediction in the masked
    # language modeling task
    random.shuffle(candidate_pred_positions)
    for mlm_pred_position in candidate_pred_positions:
        if len(pred_positions_and_labels) >= num_mlm_preds:
            break
        masked_token = None
        # 80% of the time: replace the word with the '<mask>' token
        if random.random() < 0.8:
            masked_token = '<mask>'
        else:
            # 10% of the time: keep the word unchanged
            if random.random() < 0.5:
                masked_token = tokens[mlm_pred_position]
            # 10% of the time: replace the word with a random word
            else:
                masked_token = random.choice(vocab.idx_to_token)
        mlm_input_tokens[mlm_pred_position] = masked_token
        pred_positions_and_labels.append(
            (mlm_pred_position, tokens[mlm_pred_position]))
    return mlm_input_tokens, pred_positions_and_labels
```

```
#@save
def _get_mlm_data_from_tokens(tokens, vocab):
    candidate_pred_positions = []
    # `tokens` is a list of strings
```

(continues on next page)

(continued from previous page)

```
    for i, token in enumerate(tokens):
        # Special tokens are not predicted in the masked language modeling
        # task
        if token in ['<cls>', '<sep>']:
            continue
        candidate_pred_positions.append(i)
    # 15% of random tokens are predicted in the masked language modeling task
    num_mlm_preds = max(1, round(len(tokens) * 0.15))
    mlm_input_tokens, pred_positions_and_labels = _replace_mlm_tokens(
        tokens, candidate_pred_positions, num_mlm_preds, vocab)
    pred_positions_and_labels = sorted(pred_positions_and_labels,
                                       key=lambda x: x[0])
    pred_positions = [v[0] for v in pred_positions_and_labels]
    mlm_pred_labels = [v[1] for v in pred_positions_and_labels]
    return vocab[mlm_input_tokens], pred_positions, vocab[mlm_pred_labels]
```

```
#@save
def _get_mlm_data_from_tokens(tokens, vocab):
    candidate_pred_positions = []
    # `tokens` is a list of strings
    for i, token in enumerate(tokens):
        # Special tokens are not predicted in the masked language modeling
        # task
        if token in ['<cls>', '<sep>']:
            continue
        candidate_pred_positions.append(i)
    # 15% of random tokens are predicted in the masked language modeling task
    num_mlm_preds = max(1, round(len(tokens) * 0.15))
    mlm_input_tokens, pred_positions_and_labels = _replace_mlm_tokens(
        tokens, candidate_pred_positions, num_mlm_preds, vocab)
    pred_positions_and_labels = sorted(pred_positions_and_labels,
                                       key=lambda x: x[0])
    pred_positions = [v[0] for v in pred_positions_and_labels]
    mlm_pred_labels = [v[1] for v in pred_positions_and_labels]
    return vocab[mlm_input_tokens], pred_positions, vocab[mlm_pred_labels]
```

前述の `_replace_mlm_tokens` 関数を呼び出すことで、次の関数は BERT 入力系列（`tokens`）を入力として受け取り、入力トークンのインデックス（Section 18.8.5 で説明したようなトークン置換後）、予測が行われるトークンインデックス、およびそれらのラベルインデックスを返す。

```
#@save
def _get_mlm_data_from_tokens(tokens, vocab):
    candidate_pred_positions = []
    # `tokens` is a list of strings
    for i, token in enumerate(tokens):
        # Special tokens are not predicted in the masked language modeling
        # task
        if token in ['<cls>', '<sep>']:
            continue
        candidate_pred_positions.append(i)
    # 15% of random tokens are predicted in the masked language modeling task
```

(continues on next page)

(continued from previous page)

```
    num_mlm_preds = max(1, round(len(tokens) * 0.15))
    mlm_input_tokens, pred_positions_and_labels = _replace_mlm_tokens(
        tokens, candidate_pred_positions, num_mlm_preds, vocab)
    pred_positions_and_labels = sorted(pred_positions_and_labels,
                                       key=lambda x: x[0])
    pred_positions = [v[0] for v in pred_positions_and_labels]
    mlm_pred_labels = [v[1] for v in pred_positions_and_labels]
    return vocab[mlm_input_tokens], pred_positions, vocab[mlm_pred_labels]
```

```
#@save
def _get_mlm_data_from_tokens(tokens, vocab):
    candidate_pred_positions = []
    # `tokens` is a list of strings
    for i, token in enumerate(tokens):
        # Special tokens are not predicted in the masked language modeling
        # task
        if token in ['<cls>', '<sep>']:
            continue
        candidate_pred_positions.append(i)
    # 15% of random tokens are predicted in the masked language modeling task
    num_mlm_preds = max(1, round(len(tokens) * 0.15))
    mlm_input_tokens, pred_positions_and_labels = _replace_mlm_tokens(
        tokens, candidate_pred_positions, num_mlm_preds, vocab)
    pred_positions_and_labels = sorted(pred_positions_and_labels,
                                       key=lambda x: x[0])
    pred_positions = [v[0] for v in pred_positions_and_labels]
    mlm_pred_labels = [v[1] for v in pred_positions_and_labels]
    return vocab[mlm_input_tokens], pred_positions, vocab[mlm_pred_labels]
```

```
batch_size, max_len = 512, 64
train_iter, vocab = load_data_wiki(batch_size, max_len)

for (tokens_X, segments_X, valid_lens_x, pred_positions_X, mlm_weights_X,
     mlm_Y, nsp_y) in train_iter:
    print(tokens_X.shape, segments_X.shape, valid_lens_x.shape,
          pred_positions_X.shape, mlm_weights_X.shape, mlm_Y.shape,
          nsp_y.shape)
    break
```

```
batch_size, max_len = 512, 64
train_iter, vocab = load_data_wiki(batch_size, max_len)

for (tokens_X, segments_X, valid_lens_x, pred_positions_X, mlm_weights_X,
     mlm_Y, nsp_y) in train_iter:
    print(tokens_X.shape, segments_X.shape, valid_lens_x.shape,
          pred_positions_X.shape, mlm_weights_X.shape, mlm_Y.shape,
          nsp_y.shape)
    break
```

18.9.2 テキストを事前学習データセットへ変換する

これで、BERT 事前学習用の Dataset クラスをカスタマイズする準備がほぼ整った。その前に、入力に特殊な“<pad>”トークンを追加するための補助関数 _pad_bert_inputs を定義する必要がある。その引数 examples には、2 つの事前学習タスクに対する補助関数 _get_nsp_data_from_paragraph と _get_mlm_data_from_tokens の出力が含まれる。

```
#@save
def _pad_bert_inputs(examples, max_len, vocab):
    max_num_mlm_preds = round(max_len * 0.15)
    all_token_ids, all_segments, valid_lens,  = [], [], []
    all_pred_positions, all_mlm_weights, all_mlm_labels = [], [], []
    nsp_labels = []
    for (token_ids, pred_positions, mlm_pred_label_ids, segments,
         is_next) in examples:
        all_token_ids.append(torch.tensor(token_ids + [vocab['<pad>']] * (
            max_len - len(token_ids)), dtype=torch.long))
        all_segments.append(torch.tensor(segments + [0] * (
            max_len - len(segments)), dtype=torch.long))
        # `valid_lens` excludes count of '<pad>' tokens
        valid_lens.append(torch.tensor(len(token_ids), dtype=torch.float32))
        all_pred_positions.append(torch.tensor(pred_positions + [0] * (
            max_num_mlm_preds - len(pred_positions)), dtype=torch.long))
        # Predictions of padded tokens will be filtered out in the loss via
        # multiplication of 0 weights
        all_mlm_weights.append(
            torch.tensor([1.0] * len(mlm_pred_label_ids) + [0.0] * (
                max_num_mlm_preds - len(pred_positions)),
                dtype=torch.float32))
        all_mlm_labels.append(torch.tensor(mlm_pred_label_ids + [0] * (
            max_num_mlm_preds - len(mlm_pred_label_ids)), dtype=torch.long))
        nsp_labels.append(torch.tensor(is_next, dtype=torch.long))
    return (all_token_ids, all_segments, valid_lens, all_pred_positions,
            all_mlm_weights, all_mlm_labels, nsp_labels)
```

```
#@save
def _pad_bert_inputs(examples, max_len, vocab):
    max_num_mlm_preds = round(max_len * 0.15)
    all_token_ids, all_segments, valid_lens,  = [], [], []
    all_pred_positions, all_mlm_weights, all_mlm_labels = [], [], []
    nsp_labels = []
    for (token_ids, pred_positions, mlm_pred_label_ids, segments,
         is_next) in examples:
        all_token_ids.append(np.array(token_ids + [vocab['<pad>']] * (
            max_len - len(token_ids)), dtype='int32'))
        all_segments.append(np.array(segments + [0] * (
            max_len - len(segments)), dtype='int32'))
        # `valid_lens` excludes count of '<pad>' tokens
        valid_lens.append(np.array(len(token_ids), dtype='float32'))
        all_pred_positions.append(np.array(pred_positions + [0] * (
            max_num_mlm_preds - len(pred_positions)), dtype='int32'))
        # Predictions of padded tokens will be filtered out in the loss via
        # multiplication of 0 weights
        all_mlm_weights.append(
```

(continues on next page)

(continued from previous page)

```
            np.array([1.0] * len(mlm_pred_label_ids) + [0.0] * (
                max_num_mlm_preds - len(pred_positions)), dtype='float32'))
        all_mlm_labels.append(np.array(mlm_pred_label_ids + [0] * (
            max_num_mlm_preds - len(mlm_pred_label_ids)), dtype='int32'))
        nsp_labels.append(np.array(is_next))
    return (all_token_ids, all_segments, valid_lens, all_pred_positions,
            all_mlm_weights, all_mlm_labels, nsp_labels)
```

```
len(vocab)
```

```
len(vocab)
```

2 つの事前学習タスクの学習例を生成する補助関数と、入力をパディングする補助関数をまとめて、以下の `_WikiTextDataset` クラスを BERT 事前学習用の WikiText-2 データセットとしてカスタマイズする。`__getitem__` 関数を実装することで、WikiText-2 コーパス中の文のペアから生成された事前学習（マスク付き言語モデルと次文予測）の例に任意にアクセスできる。

元の BERT モデルは、語彙サイズが 30000 の WordPiece 埋め込みを使用する (Wu *et al.*, 2016)。WordPiece のトークン化手法は、Section 18.6.2 にある元の byte pair encoding アルゴリズムをわずかに修正したものです。簡単のため、ここではトークン化に `d2l.tokenize` 関数を使う。5 回未満しか現れない低頻度トークンは除外する。

```
#@save
class _WikiTextDataset(torch.utils.data.Dataset):
    def __init__(self, paragraphs, max_len):
        # Input `paragraphs[i]` is a list of sentence strings representing a
        # paragraph; while output `paragraphs[i]` is a list of sentences
        # representing a paragraph, where each sentence is a list of tokens
        paragraphs = [d2l.tokenize(
            paragraph, token='word') for paragraph in paragraphs]
        sentences = [sentence for paragraph in paragraphs
                     for sentence in paragraph]
        self.vocab = d2l.Vocab(sentences, min_freq=5, reserved_tokens=[
            '<pad>', '<mask>', '<cls>', '<sep>'])
        # Get data for the next sentence prediction task
        examples = []
        for paragraph in paragraphs:
            examples.extend(_get_nsp_data_from_paragraph(
                paragraph, paragraphs, self.vocab, max_len))
        # Get data for the masked language model task
        examples = [(_get_mlm_data_from_tokens(tokens, self.vocab)
                      + (segments, is_next))
                     for tokens, segments, is_next in examples]
        # Pad inputs
        (self.all_token_ids, self.all_segments, self.valid_lens,
         self.all_pred_positions, self.all_mlm_weights,
         self.all_mlm_labels, self.nsp_labels) = _pad_bert_inputs(
```

(continues on next page)

(continued from previous page)

```
            examples, max_len, self.vocab)

    def __getitem__(self, idx):
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx], self.all_pred_positions[idx],
                self.all_mlm_weights[idx], self.all_mlm_labels[idx],
                self.nsp_labels[idx])

    def __len__(self):
        return len(self.all_token_ids)
```

```
#@save
class _WikiTextDataset(gluon.data.Dataset):
    def __init__(self, paragraphs, max_len):
        # Input `paragraphs[i]` is a list of sentence strings representing a
        # paragraph; while output `paragraphs[i]` is a list of sentences
        # representing a paragraph, where each sentence is a list of tokens
        paragraphs = [d2l.tokenize(
            paragraph, token='word') for paragraph in paragraphs]
        sentences = [sentence for paragraph in paragraphs
                     for sentence in paragraph]
        self.vocab = d2l.Vocab(sentences, min_freq=5, reserved_tokens=[
            '<pad>', '<mask>', '<cls>', '<sep>'])
        # Get data for the next sentence prediction task
        examples = []
        for paragraph in paragraphs:
            examples.extend(_get_nsp_data_from_paragraph(
                paragraph, paragraphs, self.vocab, max_len))
        # Get data for the masked language model task
        examples = [(_get_mlm_data_from_tokens(tokens, self.vocab)
                      + (segments, is_next))
                     for tokens, segments, is_next in examples]
        # Pad inputs
        (self.all_token_ids, self.all_segments, self.valid_lens,
         self.all_pred_positions, self.all_mlm_weights,
         self.all_mlm_labels, self.nsp_labels) = _pad_bert_inputs(
            examples, max_len, self.vocab)

    def __getitem__(self, idx):
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx], self.all_pred_positions[idx],
                self.all_mlm_weights[idx], self.all_mlm_labels[idx],
                self.nsp_labels[idx])

    def __len__(self):
        return len(self.all_token_ids)
```

`_read_wiki` 関数と `_WikiTextDataset` クラスを用いて、以下の `load_data_wiki` を定義し、WikiText-2 データセットをダウンロードして、そこから事前学習例を生成する。

```
#@save
```

(continues on next page)

(continued from previous page)

```
def load_data_wiki(batch_size, max_len):
    """Load the WikiText-2 dataset."""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('wikitext-2', 'wikitext-2')
    paragraphs = _read_wiki(data_dir)
    train_set = _WikiTextDataset(paragraphs, max_len)
    train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                        shuffle=True, num_workers=num_workers)
    return train_iter, train_set.vocab
```

```
#@save
def load_data_wiki(batch_size, max_len):
    """Load the WikiText-2 dataset."""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('wikitext-2', 'wikitext-2')
    paragraphs = _read_wiki(data_dir)
    train_set = _WikiTextDataset(paragraphs, max_len)
    train_iter = gluon.data.DataLoader(train_set, batch_size, shuffle=True,
                                       num_workers=num_workers)
    return train_iter, train_set.vocab
```

バッチサイズを 512、BERT 入力系列の最大長を 64 に設定して、BERT 事前学習例のミニバッチの形状を出力してみる。各 BERT 入力系列では、マスク付き言語モデルタスクのために 10（64×0.15）個の位置が予測対象になる。

```
batch_size, max_len = 512, 64
train_iter, vocab = load_data_wiki(batch_size, max_len)

for (tokens_X, segments_X, valid_lens_x, pred_positions_X, mlm_weights_X,
     mlm_Y, nsp_y) in train_iter:
    print(tokens_X.shape, segments_X.shape, valid_lens_x.shape,
          pred_positions_X.shape, mlm_weights_X.shape, mlm_Y.shape,
          nsp_y.shape)
    break
```

```
Downloading ../data/wikitext-2-v1.zip from https://s3.amazonaws.com/research.
↪metamind.io/wikitext/wikitext-2-v1.zip...
torch.Size([512, 64]) torch.Size([512, 64]) torch.Size([512]) torch.Size([512,␣
↪10]) torch.Size([512, 10]) torch.Size([512, 10]) torch.Size([512])
```

最後に、語彙サイズを見てみよう。低頻度トークンを除外した後でも、それは依然として PTB データセットの 2 倍以上の大きさである。

```
len(vocab)
```

```
20256
```

18.9.3 まとめ

- PTB データセットと比べると、WikiText-2 データセットは元の句読点、大文字小文字、数値を保持しており、2 倍以上大きいである。
- WikiText-2 コーパス中の文のペアから生成された事前学習（マスク付き言語モデルと次文予測）の例に任意にアクセスできる。

18.9.4 演習

1. 簡単のため、文の分割にはピリオドのみを区切り文字として使った。spaCy や NLTK など、他の文分割手法も試してみよ。例として NLTK を使う。まず NLTK をインストールする必要がある: `pip install nltk`。コードでは最初に `import nltk` する。次に、Punkt 文トークナイザをダウンロードする: `nltk.download('punkt')`。`sentences = 'This is great ! Why not ?'` のような文を分割するには、`nltk.tokenize.sent_tokenize(sentences)` を呼び出すと、2 つの文文字列からなるリスト `['This is great !', 'Why not ?']` が返りる。
2. 低頻度トークンを一切除外しない場合、語彙サイズはいくつになるか？

18.10 BERT の事前学習

Section 18.8 で実装した BERT モデルと、Section 18.9 で生成した WikiText-2 データセット由来の事前学習用サンプルを用いて、この節では WikiText-2 データセット上で BERT を事前学習する。

```
from d2l import torch as d2l
import torch
from torch import nn
```

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx

npx.set_np()
```

```
batch_size, max_len = 512, 64
train_iter, vocab = d2l.load_data_wiki(batch_size, max_len)
```


```
batch_size, max_len = 512, 64
train_iter, vocab = d2l.load_data_wiki(batch_size, max_len)
```


まず、WikiText-2 データセットを、マスク付き言語モデルと次文予測のための事前学習サンプルのミニバッチとして読み込みる。バッチサイズは 512 で、BERT

入力系列の最大長は 64 である。なお、元の BERT モデルでは最大長は 512 である。

```
batch_size, max_len = 512, 64
train_iter, vocab = d2l.load_data_wiki(batch_size, max_len)
```

```
batch_size, max_len = 512, 64
train_iter, vocab = d2l.load_data_wiki(batch_size, max_len)
```

```
Downloading ../data/wikitext-2-v1.zip from https://s3.amazonaws.com/research.
↪metamind.io/wikitext/wikitext-2-v1.zip...
[06:58:35] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
train_bert(train_iter, net, loss, len(vocab), devices, 50)
```

```
train_bert(train_iter, net, loss, len(vocab), devices, 50)
```

18.10.1 BERT の事前学習

元の BERT には、異なるモデルサイズの 2 つのバージョンがある (Devlin *et al.*, 2018)。ベースモデル（$\text{BERT}_{\text{BASE}}$）は、12 層（Transformer エンコーダブロック）を用い、768 個の隠れユニット（隠れサイズ）と 12 個の自己注意ヘッドを持つ。ラージモデル（$\text{BERT}_{\text{LARGE}}$）は、24 層、1024 個の隠れユニット、16 個の自己注意ヘッドを持つ。特に、前者は 1 億 1000 万個のパラメータを持ち、後者は 3 億 4000 万個のパラメータを持つ。デモを容易にするため、2 層、128 個の隠れユニット、2 個の自己注意ヘッドを用いた小さな BERT を定義する。

```
net = d2l.BERTModel(len(vocab), num_hiddens=128,
                    ffn_num_hiddens=256, num_heads=2, num_blks=2, dropout=0.2)
devices = d2l.try_all_gpus()
loss = nn.CrossEntropyLoss()
```

```
net = d2l.BERTModel(len(vocab), num_hiddens=128, ffn_num_hiddens=256,
                    num_heads=2, num_blks=2, dropout=0.2)
devices = d2l.try_all_gpus()
net.initialize(init.Xavier(), ctx=devices)
loss = gluon.loss.SoftmaxCELoss()
```

```
[06:59:57] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
[06:59:57] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

```
tokens_a = ['a', 'crane', 'is', 'flying']
encoded_text = get_bert_encoding(net, tokens_a)
# Tokens: '<cls>', 'a', 'crane', 'is', 'flying', '<sep>'
encoded_text_cls = encoded_text[:, 0, :]
encoded_text_crane = encoded_text[:, 2, :]
encoded_text.shape, encoded_text_cls.shape, encoded_text_crane[0][:3]
```


```
tokens_a = ['a', 'crane', 'is', 'flying']
encoded_text = get_bert_encoding(net, tokens_a)
# Tokens: '<cls>', 'a', 'crane', 'is', 'flying', '<sep>'
encoded_text_cls = encoded_text[:, 0, :]
encoded_text_crane = encoded_text[:, 2, :]
encoded_text.shape, encoded_text_cls.shape, encoded_text_crane[0][:3]
```


学習ループを定義する前に、補助関数 _get_batch_loss_bert を定義する。学習サンプルのシャードが与えられると、この関数はマスク付き言語モデルと次文予測の両方のタスクに対する損失を計算する。BERT の事前学習における最終的な損失は、マスク付き言語モデルの損失と次文予測の損失の単純な和である。

```
#@save
def _get_batch_loss_bert(net, loss, vocab_size, tokens_X,
                         segments_X, valid_lens_x,
                         pred_positions_X, mlm_weights_X,
                         mlm_Y, nsp_y):
    # Forward pass
    _, mlm_Y_hat, nsp_Y_hat = net(tokens_X, segments_X,
                                  valid_lens_x.reshape(-1),
                                  pred_positions_X)
    # Compute masked language model loss
    mlm_l = loss(mlm_Y_hat.reshape(-1, vocab_size), mlm_Y.reshape(-1)) *\
    mlm_weights_X.reshape(-1, 1)
    mlm_l = mlm_l.sum() / (mlm_weights_X.sum() + 1e-8)
    # Compute next sentence prediction loss
    nsp_l = loss(nsp_Y_hat, nsp_y)
    l = mlm_l + nsp_l
    return mlm_l, nsp_l, l
```

```
#@save
def _get_batch_loss_bert(net, loss, vocab_size, tokens_X_shards,
                         segments_X_shards, valid_lens_x_shards,
                         pred_positions_X_shards, mlm_weights_X_shards,
                         mlm_Y_shards, nsp_y_shards):
    mlm_ls, nsp_ls, ls = [], [], []
    for (tokens_X_shard, segments_X_shard, valid_lens_x_shard,
         pred_positions_X_shard, mlm_weights_X_shard, mlm_Y_shard,
         nsp_y_shard) in zip(
        tokens_X_shards, segments_X_shards, valid_lens_x_shards,
        pred_positions_X_shards, mlm_weights_X_shards, mlm_Y_shards,
        nsp_y_shards):
        # Forward pass
```

(continues on next page)

(continued from previous page)

```
        _, mlm_Y_hat, nsp_Y_hat = net(
            tokens_X_shard, segments_X_shard, valid_lens_x_shard.reshape(-1),
            pred_positions_X_shard)
        # Compute masked language model loss
        mlm_l = loss(
            mlm_Y_hat.reshape((-1, vocab_size)), mlm_Y_shard.reshape(-1),
            mlm_weights_X_shard.reshape((-1, 1)))
        mlm_l = mlm_l.sum() / (mlm_weights_X_shard.sum() + 1e-8)
        # Compute next sentence prediction loss
        nsp_l = loss(nsp_Y_hat, nsp_y_shard)
        nsp_l = nsp_l.mean()
        mlm_ls.append(mlm_l)
        nsp_ls.append(nsp_l)
        ls.append(mlm_l + nsp_l)
        npx.waitall()
    return mlm_ls, nsp_ls, ls
```

```
tokens_a, tokens_b = ['a', 'crane', 'driver', 'came'], ['he', 'just', 'left']
encoded_pair = get_bert_encoding(net, tokens_a, tokens_b)
# Tokens: '<cls>', 'a', 'crane', 'driver', 'came', '<sep>', 'he', 'just',
# 'left', '<sep>'
encoded_pair_cls = encoded_pair[:, 0, :]
encoded_pair_crane = encoded_pair[:, 2, :]
encoded_pair.shape, encoded_pair_cls.shape, encoded_pair_crane[0][:3]
```

```
tokens_a, tokens_b = ['a', 'crane', 'driver', 'came'], ['he', 'just', 'left']
encoded_pair = get_bert_encoding(net, tokens_a, tokens_b)
# Tokens: '<cls>', 'a', 'crane', 'driver', 'came', '<sep>', 'he', 'just',
# 'left', '<sep>'
encoded_pair_cls = encoded_pair[:, 0, :]
encoded_pair_crane = encoded_pair[:, 2, :]
encoded_pair.shape, encoded_pair_cls.shape, encoded_pair_crane[0][:3]
```

前述の 2 つの補助関数を呼び出して、次の train_bert 関数は、WikiText-2 (train_iter) データセット上で BERT (net) を事前学習する手順を定義する。BERT の学習には非常に長い時間がかかることがある。train_ch13 関数（Section 17.1 を参照）のように学習エポック数を指定する代わりに、以下の関数の入力 num_steps は学習の反復ステップ数を指定する。

```
def train_bert(train_iter, net, loss, vocab_size, devices, num_steps):
    net(*next(iter(train_iter))[:4])
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    trainer = torch.optim.Adam(net.parameters(), lr=0.01)
    step, timer = 0, d2l.Timer()
    animator = d2l.Animator(xlabel='step', ylabel='loss',
                            xlim=[1, num_steps], legend=['mlm', 'nsp'])
    # Sum of masked language modeling losses, sum of next sentence prediction
    # losses, no. of sentence pairs, count
    metric = d2l.Accumulator(4)
    num_steps_reached = False
```

(continues on next page)

(continued from previous page)

```
    while step < num_steps and not num_steps_reached:
        for tokens_X, segments_X, valid_lens_x, pred_positions_X,\
            mlm_weights_X, mlm_Y, nsp_y in train_iter:
            tokens_X = tokens_X.to(devices[0])
            segments_X = segments_X.to(devices[0])
            valid_lens_x = valid_lens_x.to(devices[0])
            pred_positions_X = pred_positions_X.to(devices[0])
            mlm_weights_X = mlm_weights_X.to(devices[0])
            mlm_Y, nsp_y = mlm_Y.to(devices[0]), nsp_y.to(devices[0])
            trainer.zero_grad()
            timer.start()
            mlm_l, nsp_l, l = _get_batch_loss_bert(
                net, loss, vocab_size, tokens_X, segments_X, valid_lens_x,
                pred_positions_X, mlm_weights_X, mlm_Y, nsp_y)
            l.backward()
            trainer.step()
            metric.add(mlm_l, nsp_l, tokens_X.shape[0], 1)
            timer.stop()
            animator.add(step + 1,
                         (metric[0] / metric[3], metric[1] / metric[3]))
            step += 1
            if step == num_steps:
                num_steps_reached = True
                break

    print(f'MLM loss {metric[0] / metric[3]:.3f}, '
          f'NSP loss {metric[1] / metric[3]:.3f}')
    print(f'{metric[2] / timer.sum():.1f} sentence pairs/sec on '
          f'{str(devices)}')
```

```
def train_bert(train_iter, net, loss, vocab_size, devices, num_steps):
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': 0.01})
    step, timer = 0, d2l.Timer()
    animator = d2l.Animator(xlabel='step', ylabel='loss',
                            xlim=[1, num_steps], legend=['mlm', 'nsp'])
    # Sum of masked language modeling losses, sum of next sentence prediction
    # losses, no. of sentence pairs, count
    metric = d2l.Accumulator(4)
    num_steps_reached = False
    while step < num_steps and not num_steps_reached:
        for batch in train_iter:
            (tokens_X_shards, segments_X_shards, valid_lens_x_shards,
             pred_positions_X_shards, mlm_weights_X_shards,
             mlm_Y_shards, nsp_y_shards) = [gluon.utils.split_and_load(
                elem, devices, even_split=False) for elem in batch]
            timer.start()
            with autograd.record():
                mlm_ls, nsp_ls, ls = _get_batch_loss_bert(
                    net, loss, vocab_size, tokens_X_shards, segments_X_shards,
                    valid_lens_x_shards, pred_positions_X_shards,
                    mlm_weights_X_shards, mlm_Y_shards, nsp_y_shards)
            for l in ls:
                l.backward()
```

(continues on next page)

(continued from previous page)

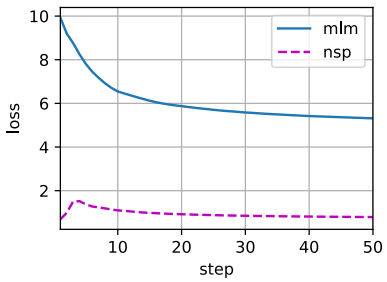
```
            trainer.step(1)
            mlm_l_mean = sum([float(l) for l in mlm_ls]) / len(mlm_ls)
            nsp_l_mean = sum([float(l) for l in nsp_ls]) / len(nsp_ls)
            metric.add(mlm_l_mean, nsp_l_mean, batch[0].shape[0], 1)
            timer.stop()
            animator.add(step + 1,
                         (metric[0] / metric[3], metric[1] / metric[3]))
            step += 1
            if step == num_steps:
                num_steps_reached = True
                break

    print(f'MLM loss {metric[0] / metric[3]:.3f}, '
          f'NSP loss {metric[1] / metric[3]:.3f}')
    print(f'{metric[2] / timer.sum():.1f} sentence pairs/sec on '
          f'{str(devices)}')
```

BERT の事前学習中に、マスク付き言語モデルの損失と次文予測の損失の両方をプロットできる。

```
train_bert(train_iter, net, loss, len(vocab), devices, 50)
```

```
MLM loss 5.315, NSP loss 0.790
4179.6 sentence pairs/sec on [device(type='cuda', index=0), device(type='cuda',
↪ index=1)]
```



18.10.2 BERT によるテキスト表現

BERT を事前学習した後は、単一のテキスト、テキストのペア、あるいはそれらの中の任意のトークンを表現するために利用できる。次の関数は、tokens_a と tokens_b に含まれるすべてのトークンに対する BERT (net) の表現を返す。

```
def get_bert_encoding(net, tokens_a, tokens_b=None):
    tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
    token_ids = torch.tensor(vocab[tokens], device=devices[0]).unsqueeze(0)
    segments = torch.tensor(segments, device=devices[0]).unsqueeze(0)
```

(continues on next page)

(continued from previous page)

```
    valid_len = torch.tensor(len(tokens), device=devices[0]).unsqueeze(0)
    encoded_X, _, _ = net(token_ids, segments, valid_len)
    return encoded_X
```

```
def get_bert_encoding(net, tokens_a, tokens_b=None):
    tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
    token_ids = np.expand_dims(np.array(vocab[tokens], ctx=devices[0]),
                               axis=0)
    segments = np.expand_dims(np.array(segments, ctx=devices[0]), axis=0)
    valid_len = np.expand_dims(np.array(len(tokens), ctx=devices[0]), axis=0)
    encoded_X, _, _ = net(token_ids, segments, valid_len)
    return encoded_X
```

文“a crane is flying”を考える。Section 18.8.4 で説明した BERT の入力表現を思い出してほしい。特殊トークン“<cls>”（分類に使用）と“<sep>”（区切りに使用）を挿入すると、BERT の入力系列の長さは 6 になる。“<cls>”トークンのインデックスは 0 なので、`encoded_text[:, 0, :]` は入力文全体の BERT 表現である。多義語トークン“crane”を評価するために、その BERT 表現の最初の 3 要素も出力する。

```
tokens_a = ['a', 'crane', 'is', 'flying']
encoded_text = get_bert_encoding(net, tokens_a)
# Tokens: '<cls>', 'a', 'crane', 'is', 'flying', '<sep>'
encoded_text_cls = encoded_text[:, 0, :]
encoded_text_crane = encoded_text[:, 2, :]
encoded_text.shape, encoded_text_cls.shape, encoded_text_crane[0][:3]
```

```
(torch.Size([1, 6, 128]),
 torch.Size([1, 128]),
 tensor([-0.7035,  1.4329, -0.3273], device='cuda:0', grad_fn=<SliceBackward0>
↪))
```

次に、文対“a crane driver came”と“he just left”を考える。同様に、`encoded_pair[:, 0, :]` は事前学習済み BERT による文対全体の符号化結果である。多義語トークン“crane”の最初の 3 要素は、文脈が異なるときには先ほどとは異なることに注意しよ。これは、BERT の表現が文脈依存であることを示している。

```
tokens_a, tokens_b = ['a', 'crane', 'driver', 'came'], ['he', 'just', 'left']
encoded_pair = get_bert_encoding(net, tokens_a, tokens_b)
# Tokens: '<cls>', 'a', 'crane', 'driver', 'came', '<sep>', 'he', 'just',
# 'left', '<sep>'
encoded_pair_cls = encoded_pair[:, 0, :]
encoded_pair_crane = encoded_pair[:, 2, :]
encoded_pair.shape, encoded_pair_cls.shape, encoded_pair_crane[0][:3]
```

```
(torch.Size([1, 10, 128]),
 torch.Size([1, 128]),
```

(continues on next page)

(continued from previous page)

```
tensor([ 0.0862,  0.0970, -0.2512], device='cuda:0', grad_fn=<SliceBackward0>
↪))
```

Section 19 では、事前学習済み BERT モデルを下流の自然言語処理アプリケーション向けにファインチューニングする。

18.10.3 要約

- 元の BERT には 2 つのバージョンがあり、ベースモデルは 1 億 1000 万個のパラメータを持ち、ラージモデルは 3 億 4000 万個のパラメータを持つ。
- BERT を事前学習した後は、単一テキスト、テキスト対、あるいはそれらの中の任意のトークンを表現するために利用できる。
- 実験では、同じトークンでも文脈が異なると BERT 表現が異なる。これは、BERT の表現が文脈依存であることを示している。

18.10.4 演習

1. 実験から、マスク付き言語モデルの損失が次文予測の損失よりかなり大きいことがわかる。なぜだろうか。
2. BERT 入力系列の最大長を 512（元の BERT モデルと同じ）に設定する。元の BERT モデル、たとえば $\text{BERT}_{\text{LARGE}}$ の設定を使っよ。この節を実行するとエラーは発生するか。なぜだろうか。

19 自然言語処理: 応用

私たちは、テキスト系列中のトークンをどのように表現し、Section 18 でその表現をどのように学習するかを見てきた。
このような事前学習済みのテキスト表現は、さまざまな下流の自然言語処理タスクに対して、各種モデルへ入力できる。

実際、前の章では、深層学習アーキテクチャを説明するためだけに、事前学習なしで自然言語処理の応用をすでにいくつか扱ってきた。たとえば、Section 12 では、RNN を用いてノヴェラのようなテキストを生成する言語モデルを設計した。また、Section 13 と Section 14 では、機械翻訳のために RNN と注意機構に基づくモデルも設計した。

しかし、この本はそのような応用を網羅的に扱うことを目的としていない。その代わりに、私たちが注目するのは、言語の（深層）表現学習を自然言語処理の問題解決にどう適用するかである。事前学習済みのテキスト表現が与えられたとき、この章では、単一テキストとテキスト対の関係をそれぞれ解析する、2 つの人気があり代表的な下流の自然言語処理タスク、すなわち感情分析と自然言語推論を取り上げる。

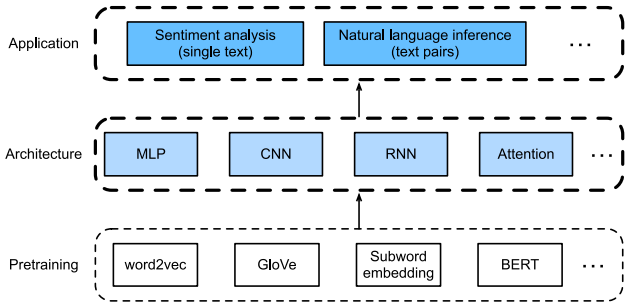


Fig. 19.1 事前学習済みのテキスト表現は、さまざまな深層学習アーキテクチャに入力して、異なる下流の自然言語処理応用に利用できる。この章では、異なる下流の自然言語処理応用のためにモデルをどのように設計するかに焦点を当てる。

Fig. 19.1 に示すように、この章では、MLP、CNN、RNN、注意機構など、さまざまな種類の深層学習アーキテクチャを用いて自然言語処理モデルを設計する基本的な考え方を説明する。Fig. 19.1 にあるどの応用に対しても、どの事前学習済みテキスト表現とどのアーキテクチャを組み合わせることも可能ですが、ここでは代表的な組み合わせをいくつか選ぶ。具体的には、感情分析について

は、RNN と CNN に基づく人気のあるアーキテクチャを調べる。自然言語推論については、テキスト対を解析する方法を示すために、注意機構と MLP を選ぶ。最後に、事前学習済み BERT モデルを幅広い自然言語処理応用に対してどのようにファインチューニングするかを紹介する。対象は、系列レベル（単一テキスト分類とテキスト対分類）とトークンレベル（テキストタグ付けと質問応答）である。具体的な実証例として、自然言語推論のために BERT をファインチューニングする。

Section 18.8 で紹介したように、BERT は幅広い自然言語処理応用に対して、アーキテクチャの変更を最小限に抑えられる。しかし、この利点には、下流応用のために BERT の膨大な数のパラメータをファインチューニングしなければならないという代償が伴いる。空間や時間に制約がある場合には、MLP、CNN、RNN、注意機構に基づく、こうした工夫されたモデルのほうがより実用的である。以下では、まず感情分析の応用から始め、それぞれ RNN と CNN に基づくモデル設計を説明する。

19.1 感情分析とデータセット

オンラインのソーシャルメディアやレビュー・プラットフォームの普及に伴い、意見を含むデータが大量に記録されるようになり、意思決定プロセスを支援するうえで大きな可能性を秘めている。感情分析は、製品レビュー、ブログコメント、フォーラムでの議論など、人々が生成したテキストに表れる感情を研究する分野である。これは、政治（たとえば政策に対する世論の分析）、金融（たとえば市場の感情の分析）、マーケティング（たとえば製品調査やブランド管理）など、非常に多様な分野で広く応用されている。

感情は正負などの離散的な極性や尺度として分類できるため、感情分析はテキスト分類タスクとみなすことができる。これは、長さが可変のテキスト系列を固定長のテキストカテゴリへと変換するものである。この章では、感情分析のために Stanford の large movie review dataset[116] を使用する。このデータセットは、IMDb からダウンロードされた 25000 件の映画レビューを含む訓練セットとテストセットから構成されている。どちらのデータセットにも、“positive”と“negative”のラベルが同数含まれており、異なる感情の極性を示している。

116



```
from d2l import torch as d2l
import torch
from torch import nn
import os
```

```
from d2l import mxnet as d2l
from mxnet import np, npx
```

(continues on next page)

(continued from previous page)

```
import os
npx.set_np()
```

```
#@save
d2l.DATA_HUB['aclImdb'] = (d2l.DATA_URL + 'aclImdb_v1.tar.gz',
                           '01ada507287d82875905620988597833ad4e0903')

data_dir = d2l.download_extract('aclImdb', 'aclImdb')
```


```
#@save
d2l.DATA_HUB['aclImdb'] = (d2l.DATA_URL + 'aclImdb_v1.tar.gz',
                           '01ada507287d82875905620988597833ad4e0903')

data_dir = d2l.download_extract('aclImdb', 'aclImdb')
```


19.1.1 データセットの読み込み

まず、この IMDb レビューデータセットを ../data/aclImdb のパスにダウンロードして展開する。


```
#@save
d2l.DATA_HUB['aclImdb'] = (d2l.DATA_URL + 'aclImdb_v1.tar.gz',
                           '01ada507287d82875905620988597833ad4e0903')

data_dir = d2l.download_extract('aclImdb', 'aclImdb')
```



```
#@save
d2l.DATA_HUB['aclImdb'] = (d2l.DATA_URL + 'aclImdb_v1.tar.gz',
                           '01ada507287d82875905620988597833ad4e0903')

data_dir = d2l.download_extract('aclImdb', 'aclImdb')
```


```
#@save
def read_imdb(data_dir, is_train):
    """IMDb レビューデータセットのテキスト系列とラベルを読み込む。"""
    data, labels = [], []
    for label in ('pos', 'neg'):
        folder_name = os.path.join(data_dir, 'train' if is_train else 'test',
                                   label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb(data_dir, is_train=True)
print('# trainings:', len(train_data[0]))
```

(continues on next page)

(continued from previous page)

```
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[:60])
```

```
#@save
def read_imdb(data_dir, is_train):
    """IMDb レビューデータセットのテキスト系列とラベルを読み込む。"""
    data, labels = [], []
    for label in ('pos', 'neg'):
        folder_name = os.path.join(data_dir, 'train' if is_train else 'test',
                                   label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb(data_dir, is_train=True)
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[:60])
```

次に、訓練データセットとテストデータセットを読み込みる。各サンプルはレビューとそのラベルであり、1 は“positive”、0 は“negative”を表す。

```
#@save
def read_imdb(data_dir, is_train):
    """IMDb レビューデータセットのテキスト系列とラベルを読み込む。"""
    data, labels = [], []
    for label in ('pos', 'neg'):
        folder_name = os.path.join(data_dir, 'train' if is_train else 'test',
                                   label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb(data_dir, is_train=True)
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[:60])
```

```
# trainings: 25000
label: 1 review: Zentropa has much in common with The Third Man, another noir
label: 1 review: Zentropa is the most original movie I've seen in years. If y
label: 1 review: Lars Von Trier is never backward in trying out new technique
```

```
#@save
```

(continues on next page)

(continued from previous page)

```
def read_imdb(data_dir, is_train):
    """IMDb レビューデータセットのテキスト系列とラベルを読み込む。"""
    data, labels = [], []
    for label in ('pos', 'neg'):
        folder_name = os.path.join(data_dir, 'train' if is_train else 'test',
                                   label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb(data_dir, is_train=True)
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[:60])
```

```
# trainings: 25000
label: 1 review: Zentropa has much in common with The Third Man, another noir
label: 1 review: Zentropa is the most original movie I've seen in years. If y
label: 1 review: Lars Von Trier is never backward in trying out new technique
```

```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])
```


```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])
```


19.1.2 データセットの前処理

各単語をトークンとして扱い、5 回未満しか出現しない単語を除外して、訓練データセットから語彙を作成する。


```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])
```



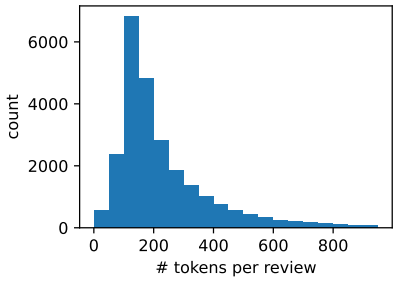
```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])
```


```
d2l.set_figsize()
d2l.plt.xlabel('# tokens per review')
d2l.plt.ylabel('count')
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
```

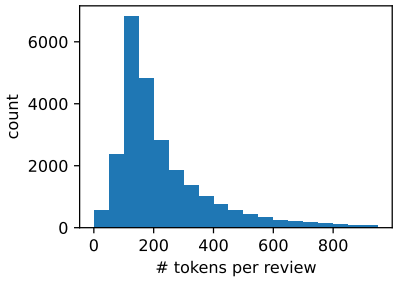
```
d2l.set_figsize()
d2l.plt.xlabel('# tokens per review')
d2l.plt.ylabel('count')
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
```

トークン化した後で、レビュー長のヒストグラムを描いてみよう。

```
d2l.set_figsize()
d2l.plt.xlabel('# tokens per review')
d2l.plt.ylabel('count')
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
```




```
d2l.set_figsize()
d2l.plt.xlabel('# tokens per review')
d2l.plt.ylabel('count')
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
```




```
num_steps = 500  # sequence length
train_features = d2l.tensor([d2l.truncate_pad(
    vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
print(train_features.shape)
```

```
num_steps = 500  # sequence length
train_features = d2l.tensor([d2l.truncate_pad(
    vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
print(train_features.shape)
```

予想どおり、レビューの長さはさまざまである。このようなレビューのミニバッチを一度に処理するために、切り詰めとパディングを用いて各レビューの長さを 500 に設定する。これは、Section 13.5 における機械翻訳データセットの前処理手順と似ている。

```
num_steps = 500  # sequence length
train_features = d2l.tensor([d2l.truncate_pad(
    vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
print(train_features.shape)
```

```
torch.Size([25000, 500])
```

19.1.3 データイテレータの作成

これでデータイテレータを作成できる。各反復で、サンプルのミニバッチが返される。

```
train_iter = d2l.load_array((train_features, torch.tensor(train_data[1])), 64)

for X, y in train_iter:
    print('X:', X.shape, ', y:', y.shape)
    break
print('# batches:', len(train_iter))
```

```
X: torch.Size([64, 500]) , y: torch.Size([64])
# batches: 391
```

```
train_iter = d2l.load_array((train_features, train_data[1]), 64)

for X, y in train_iter:
    print('X:', X.shape, ', y:', y.shape)
    break
print('# batches:', len(train_iter))
```

```
X: (64, 500) , y: (64,)
# batches: 391
```

19.1.4 まとめて実行する

最後に、上記の手順を `load_data_imdb` 関数にまとめる。この関数は、訓練データとテストデータのイテレータ、および IMDb レビューデータセットの語彙を返す。

```
#@save
def load_data_imdb(batch_size, num_steps=500):
```

(continues on next page)

(continued from previous page)

```
    """IMDb レビューデータセットのデータイテレータと語彙を返す。"""
    data_dir = d2l.download_extract('aclImdb', 'aclImdb')
    train_data = read_imdb(data_dir, True)
    test_data = read_imdb(data_dir, False)
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')
    vocab = d2l.Vocab(train_tokens, min_freq=5)
    train_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
    test_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in test_tokens])
    train_iter = d2l.load_array((train_features, torch.tensor(train_data[1])),
                                batch_size)
    test_iter = d2l.load_array((test_features, torch.tensor(test_data[1])),
                               batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

```
#@save
def load_data_imdb(batch_size, num_steps=500):
    """IMDb レビューデータセットのデータイテレータと語彙を返す。"""
    data_dir = d2l.download_extract('aclImdb', 'aclImdb')
    train_data = read_imdb(data_dir, True)
    test_data = read_imdb(data_dir, False)
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')
    vocab = d2l.Vocab(train_tokens, min_freq=5)
    train_features = np.array([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
    test_features = np.array([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in test_tokens])
    train_iter = d2l.load_array((train_features, train_data[1]), batch_size)
    test_iter = d2l.load_array((test_features, test_data[1]), batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

19.1.5 まとめ

- 感情分析は、人々が生成したテキストに表れる感情を研究するものであり、長さが可変のテキスト系列を固定長のテキストカテゴリへ変換するテキスト分類問題とみなせる。
- 前処理後、Stanford の large movie review dataset（IMDb レビューデータセット）を、語彙とともにデータイテレータへ読み込むことができる。

19.1.6 演習

1. この節で、感情分析モデルの学習を高速化するために変更できるハイパーパラメータは何ですか？
2. Amazon reviews[117] のデータセットを、感情分析用のデータイテレータとラベ

117


ルに読み込む関数を実装できるか？

19.2 感情分析: 再帰型ニューラルネットワークの使用

単語類似度や類推のタスクと同様に、感情分析にも事前学習済みの単語ベクトルを適用できる。Section 19.1 の IMDb レビューデータセットはそれほど大きくないため、大規模コーパスで事前学習されたテキスト表現を用いることで、モデルの過学習を抑えられる可能性がある。Fig. 19.2.1 に示す具体例では、各トークンを事前学習済みの GloVe モデルで表現し、それらのトークン表現を多層双方向 RNN に入力してテキスト系列表現を得る。その表現は感情分析の出力へと変換される (Maas *et al.*, 2011)。同じ下流アプリケーションに対して、後ほど別のアーキテクチャ上の選択肢も検討する。

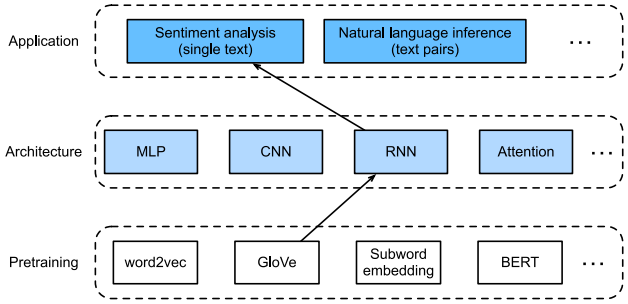


Fig. 19.2.1 この節では、事前学習済み GloVe を RNN ベースの感情分析アーキテクチャに入力する。

```
from d2l import torch as d2l
import torch
from torch import nn

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

```
from d2l import mxnet as d2l
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn, rnn
npx.set_np()

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

```
Downloading ../data/aclImdb_v1.tar.gz from http://d2l-data.s3-accelerate.
↪amazonaws.com/aclImdb_v1.tar.gz...
[07:03:42] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
embed_size, num_hiddens, num_layers, devices = 100, 100, 2, d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)
```

```
embed_size, num_hiddens, num_layers, devices = 100, 100, 2, d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)
```

19.2.1 RNN による単一テキストの表現

感情分析のようなテキスト分類タスクでは、可変長のテキスト系列を固定長のカテゴリへ変換する。以下の BiRNN クラスでは、テキスト系列の各トークンが埋め込み層 (self.embedding) を通じて個別の事前学習済み GloVe 表現を得るが、系列全体は双方向 RNN (self.encoder) によって符号化される。より具体的には、双方向 LSTM の（最終層における）最初と最後の時間ステップでの隠れ状態を連結し、テキスト系列の表現とする。この単一のテキスト表現は、全結合層 (self.decoder) によって 2 つの出力（“positive”と“negative”）を持つ出力カテゴリへ変換される。

```
class BiRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Set `bidirectional` to True to get a bidirectional RNN
        self.encoder = nn.LSTM(embed_size, num_hiddens, num_layers=num_layers,
                               bidirectional=True)
        self.decoder = nn.Linear(4 * num_hiddens, 2)

    def forward(self, inputs):
        # The shape of `inputs` is (batch size, no. of time steps). Because
        # LSTM requires its input's first dimension to be the temporal
        # dimension, the input is transposed before obtaining token
        # representations. The output shape is (no. of time steps, batch size,
        # word vector dimension)
        embeddings = self.embedding(inputs.T)
        self.encoder.flatten_parameters()
        # Returns hidden states of the last hidden layer at different time
        # steps. The shape of `outputs` is (no. of time steps, batch size,
        # 2 * no. of hidden units)
        outputs, _ = self.encoder(embeddings)
        # Concatenate the hidden states at the initial and final time steps as
        # the input of the fully connected layer. Its shape is (batch size,
        # 4 * no. of hidden units)
        encoding = torch.cat((outputs[0], outputs[-1]), dim=1)
        outs = self.decoder(encoding)
        return outs
```

```
class BiRNN(nn.Block):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Set `bidirectional` to True to get a bidirectional RNN
        self.encoder = rnn.LSTM(num_hiddens, num_layers=num_layers,
                                bidirectional=True, input_size=embed_size)
        self.decoder = nn.Dense(2)

    def forward(self, inputs):
        # The shape of `inputs` is (batch size, no. of time steps). Because
        # LSTM requires its input's first dimension to be the temporal
        # dimension, the input is transposed before obtaining token
        # representations. The output shape is (no. of time steps, batch size,
        # word vector dimension)
        embeddings = self.embedding(inputs.T)
        # Returns hidden states of the last hidden layer at different time
        # steps. The shape of `outputs` is (no. of time steps, batch size,
        # 2 * no. of hidden units)
        outputs = self.encoder(embeddings)
        # Concatenate the hidden states at the initial and final time steps as
        # the input of the fully connected layer. Its shape is (batch size,
        # 4 * no. of hidden units)
        encoding = np.concatenate((outputs[0], outputs[-1]), axis=1)
        outs = self.decoder(encoding)
        return outs
```

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
```


```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
```


感情分析のために単一テキストを表現する双方向 RNN を、2 層の隠れ層で構成してみよう。

```
embed_size, num_hiddens, num_layers, devices = 100, 100, 2, d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)
```


```
embed_size, num_hiddens, num_layers, devices = 100, 100, 2, d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)
```


```
embeds = glove_embedding[vocab.idx_to_token]
embeds.shape
```


```
embeds = glove_embedding[vocab.idx_to_token]
embeds.shape
```

```
def init_weights(module):
    if type(module) == nn.Linear:
        nn.init.xavier_uniform_(module.weight)
    if type(module) == nn.LSTM:
        for param in module._flat_weights_names:
            if "weight" in param:
                nn.init.xavier_uniform_(module._parameters[param])
net.apply(init_weights);
```

```
net.initialize(init.Xavier(), ctx=devices)
```

```
[07:03:48] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
[07:03:48] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
predict_sentiment(net, vocab, 'this movie is so great')
```

19.2.2 事前学習済み単語ベクトルの読み込み

以下では、語彙内のトークンに対応する事前学習済みの 100 次元（embed_size と一致している必要がある）の GloVe 埋め込みを読み込みる。

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
```

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
```

```
Downloading ../data/glove.6B.100d.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/glove.6B.100d.zip...
```

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

語彙内のすべてのトークンに対するベクトルの形状を表示す。

```
embeds = glove_embedding[vocab.idx_to_token]
embeds.shape
```

```
torch.Size([49346, 100])
```

これらの事前学習済み単語ベクトルを用いてレビュー中のトークンを表現し、学習中にこれらのベクトルは更新しない。

```
net.embedding.weight.data.copy_(embeds)
net.embedding.weight.requires_grad = False
```

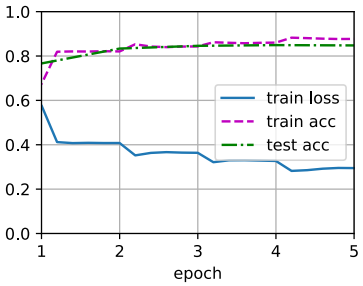
```
net.embedding.weight.set_data(embeds)
net.embedding.collect_params().setattr('grad_req', 'null')
```

19.2.3 モデルの学習と評価

これで、感情分析のために双方向 RNN を学習できる。

```
lr, num_epochs = 0.01, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

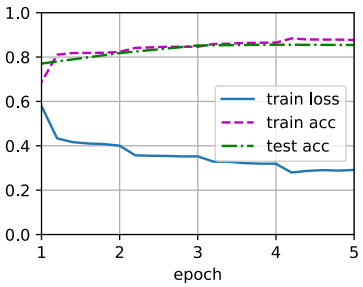
```
loss 0.295, train acc 0.877, test acc 0.848
2845.4 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```




```
lr, num_epochs = 0.01, 5
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.291, train acc 0.877, test acc 0.854
747.8 examples/sec on [gpu(0), gpu(1)]
```

学習済みモデル net を用いてテキスト系列の感情を予測するために、以下の関数を定義する。




```
#@save
def predict_sentiment(net, vocab, sequence):
    """Predict the sentiment of a text sequence."""
    sequence = torch.tensor(vocab[sequence.split()], device=d2l.try_gpu())
    label = torch.argmax(net(sequence.reshape(1, -1)), dim=1)
    return 'positive' if label == 1 else 'negative'
```

```
#@save
def predict_sentiment(net, vocab, sequence):
    """Predict the sentiment of a text sequence."""
    sequence = np.array(vocab[sequence.split()], ctx=d2l.try_gpu())
    label = np.argmax(net(sequence.reshape(1, -1)), axis=1)
    return 'positive' if label == 1 else 'negative'
```

最後に、学習済みモデルを使って 2 つの簡単な文の感情を予測してみよう。

```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

19.2.4 まとめ

- 事前学習済み単語ベクトルは、テキスト系列内の個々のトークンを表現できる。
- 双方向 RNN は、最初と最後の時間ステップでの隠れ状態の連結などによって、テキスト系列を表現できる。この単一のテキスト表現は、全結合層を用いてカテゴリへ変換できる。

19.2.5 演習

1. エポック数を増やしてみよう。学習精度とテスト精度を改善できるか？他のハイパーパラメータを調整した場合はどうだろうか？
2. 300 次元 GloVe 埋め込みのような、より大きな事前学習済み単語ベクトルを使ってみよう。分類精度は向上するか？
3. spaCy のトークン化を使うことで分類精度を改善できるか？spaCy をインストールし（`pip install spacy`）、英語パッケージをインストールする必要がある（`python -m spacy download en`）。コードでは、まず spaCy をインポートし（`import spacy`）、次に spaCy の英語パッケージを読み込みます（`spacy_en = spacy.load('en')`）。最後に、`def tokenizer(text): return [tok.text for tok in spacy_en.tokenizer(text)]` を定義して、元の `tokenizer` 関数を置き換えよ。GloVe と spaCy ではフレーズトークンの形式が異なることに注意しよ。たとえば、フレーズトークン“new york”は、GloVe では“new-york”の形式であり、spaCy のトークン化後は“new york”の形式になる。

19.3 畳み込みニューラルネットワークを用いた感情分析

Section 10 では、2 次元 CNN を用いて 2 次元画像データを処理する仕組みを調べた。そこでは、隣接ピクセルのような局所特徴に対して CNN が適用された。CNN はもともとコンピュータビジョン向けに設計されたが、自然言語処理でも広く使われている。簡単に言えば、任意のテキスト系列を 1 次元画像だと考えればよいのである。このようにすると、1 次元 CNN はテキスト中の n-gram のような局所特徴を処理できる。

この節では、単一のテキストを表現するための CNN アーキテクチャをどのように設計するかを示すために、*textCNN* モデルを用いる (Kim, 2014)。感情分析のために GloVe の事前学習を用いた RNN アーキテクチャを使う Fig. 19.2.1 と比べると、Fig. 19.3.1 における唯一の違いはアーキテクチャの選択にある。

```
from d2l import torch as d2l
import torch
from torch import nn

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

```
Downloading ../data/aclImdb_v1.tar.gz from http://d2l-data.s3-accelerate.
↪amazonaws.com/aclImdb_v1.tar.gz...
```

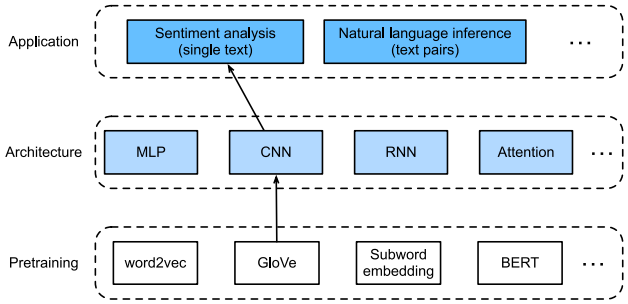


Fig. 19.3.1 この節では、事前学習済みの GloVe を CNN ベースの感情分析アーキテクチャに入力する。

```
from d2l import mxnet as d2l
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

```
[07:17:55] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
def corr1d(X, K):
    w = K.shape[0]
    Y = d2l.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i: i + w] * K).sum()
    return Y
```

```
def corr1d(X, K):
    w = K.shape[0]
    Y = d2l.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i: i + w] * K).sum()
    return Y
```

19.3.1 1 次元畳み込み

モデルを紹介する前に、1 次元畳み込みがどのように動作するかを見てみよう。これは、相互相関演算に基づく 2 次元畳み込みの特殊な場合にすぎないことを念頭に置いよ。

Fig. 19.3.2 に示すように、1 次元の場合、畳み込みウィンドウは入力テンソル上を左から右へとスライドする。スライド中、畳み込みウィンドウのある位置に含まれる入力部分テンソル（たとえば Fig. 19.3.2 の 0 と 1）とカーネルテンソ

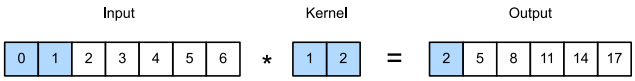


Fig. 19.3.2 1 次元の相互相関演算。網掛け部分は最初の出力要素と、その出力計算に使われる入力テンソルおよびカーネルテンソルの要素を示している: $0 \times 1 + 1 \times 2 = 2$.

ル（たとえば Fig. 19.3.2 の 1 と 2）を要素ごとに掛け合わせる。これらの積の和が、出力テンソルの対応する位置における 1 つのスカラー値（たとえば Fig. 19.3.2 の $0 \times 1 + 1 \times 2 = 2$）を与える。

以下の `corr1d` 関数で 1 次元の相互相関を実装する。入力テンソル `X` とカーネルテンソル `K` が与えられると、出力テンソル `Y` を返す。

```
def corr1d(X, K):
    w = K.shape[0]
    Y = d2l.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i: i + w] * K).sum()
    return Y
```


```
def corr1d(X, K):
    w = K.shape[0]
    Y = d2l.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i: i + w] * K).sum()
    return Y
```


```
X, K = d2l.tensor([0, 1, 2, 3, 4, 5, 6]), d2l.tensor([1, 2])
corr1d(X, K)
```


```
X, K = d2l.tensor([0, 1, 2, 3, 4, 5, 6]), d2l.tensor([1, 2])
corr1d(X, K)
```


Fig. 19.3.2 の入力テンソル `X` とカーネルテンソル `K` を構成して、上の 1 次元相互相関実装の出力を検証できる。

```
X, K = d2l.tensor([0, 1, 2, 3, 4, 5, 6]), d2l.tensor([1, 2])
corr1d(X, K)
```

```
tensor([ 2.,  5.,  8., 11., 14., 17.])
```


```
X, K = d2l.tensor([0, 1, 2, 3, 4, 5, 6]), d2l.tensor([1, 2])
corr1d(X, K)
```

```
array([ 2.,  5.,  8., 11., 14., 17.])
```

```
def corr1d_multi_in(X, K):
    # まず、`X` と `K` の 0 次元目（チャネル次元）を反復する。
    # その後、それらを足し合わせる
    return sum(corr1d(x, k) for x, k in zip(X, K))

X = d2l.tensor([[0, 1, 2, 3, 4, 5, 6],
                [1, 2, 3, 4, 5, 6, 7],
                [2, 3, 4, 5, 6, 7, 8]])
K = d2l.tensor([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)
```


```
def corr1d_multi_in(X, K):
    # まず、`X` と `K` の 0 次元目（チャネル次元）を反復する。
    # その後、それらを足し合わせる
    return sum(corr1d(x, k) for x, k in zip(X, K))

X = d2l.tensor([[0, 1, 2, 3, 4, 5, 6],
                [1, 2, 3, 4, 5, 6, 7],
                [2, 3, 4, 5, 6, 7, 8]])
K = d2l.tensor([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)
```


複数チャネルを持つ任意の 1 次元入力に対しては、畳み込みカーネルも同じ数の入力チャネルを持つ必要がある。その後、各チャネルごとに、入力の 1 次元テンソルと畳み込みカーネルの 1 次元テンソルに対して相互相関演算を行い、すべてのチャネルにわたる結果を足し合わせて 1 次元の出力テンソルを生成する。Fig. 19.3.3 は 3 つの入力チャネルを持つ 1 次元相互相関演算を示している。

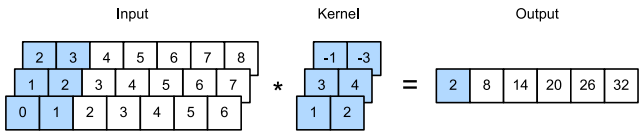


Fig. 19.3.3 3 つの入力チャネルを持つ 1 次元の相互相関演算。網掛け部分は最初の出力要素と、その出力計算に使われる入力テンソルおよびカーネルテンソルの要素を示している: $0 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 + 2 \times (-1) + 3 \times (-3) = 2$.

複数入力チャネルを持つ 1 次元相互相関演算を実装し、Fig. 19.3.3 の結果を検証できる。

```
def corr1d_multi_in(X, K):
    # まず、`X` と `K` の 0 次元目（チャネル次元）を反復する。
    # その後、それらを足し合わせる
    return sum(corr1d(x, k) for x, k in zip(X, K))
```

(continues on next page)

(continued from previous page)

```
X = d2l.tensor([[0, 1, 2, 3, 4, 5, 6],
                [1, 2, 3, 4, 5, 6, 7],
                [2, 3, 4, 5, 6, 7, 8]])
K = d2l.tensor([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)
```

```
tensor([ 2.,  8., 14., 20., 26., 32.])
```

```
def corr1d_multi_in(X, K):
    # まず、`X` と `K` の 0 次元目（チャネル次元）を反復する。
    # その後、それらを足し合わせる
    return sum(corr1d(x, k) for x, k in zip(X, K))

X = d2l.tensor([[0, 1, 2, 3, 4, 5, 6],
                [1, 2, 3, 4, 5, 6, 7],
                [2, 3, 4, 5, 6, 7, 8]])
K = d2l.tensor([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)
```

```
array([ 2.,  8., 14., 20., 26., 32.])
```

```
d2l.predict_sentiment(net, vocab, 'this movie is so great')
```


```
d2l.predict_sentiment(net, vocab, 'this movie is so great')
```


複数入力チャネルを持つ 1 次元相互相関は、単一入力チャネルの 2 次元相互相関と等価であることに注意しよ。例として、Fig. 19.3.3 における複数入力チャネルの 1 次元相互相関に対応する形は、Fig. 19.3.4 における単一入力チャネルの 2 次元相互相関である。ここでは、畳み込みカーネルの高さは入力テンソルの高さと同じでなければならない。

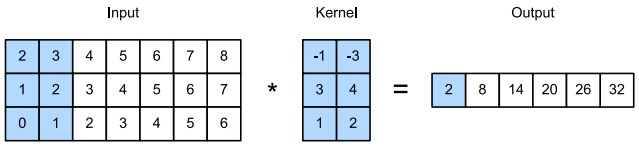


Fig. 19.3.4 単一入力チャネルを持つ 2 次元の相互相関演算。網掛け部分は最初の出力要素と、その出力計算に使われる入力テンソルおよびカーネルテンソルの要素を示している: $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$.

Fig. 19.3.2 と Fig. 19.3.3 の両方の出力は、どちらも 1 チャネルしか持たない。Section 10.4.2 で説明した複数出力チャネルを持つ 2 次元畳み込みと同様に、1 次元畳み込みでも複数の出力チャネルを指定できる。

19.3.2 時間方向最大プーリング

同様に、プーリングを用いて系列表現から各時刻にわたる最も重要な特徴として最大値を抽出できる。textCNN で使われる *max-over-time pooling* は、1 次元のグローバル最大プーリング

(Collobert *et al.*, 2011)

のように動作する。各チャネルが異なる時刻の値を保持する多チャネル入力に対しては、各チャネルの出力はそのチャネルの最大値になる。なお、max-over-time pooling ではチャネルごとに時刻数が異なっていても構わない。

19.3.3 textCNN モデル

1 次元畳み込みと max-over-time pooling を用いて、textCNN モデルは個々の事前学習済みトークン表現を入力とし、その後、下流タスク向けの系列表現を得て変換する。

n 個のトークンを d 次元ベクトルで表した単一のテキスト系列に対して、入力テンソルの幅、高さ、チャネル数はそれぞれ n、1、d である。textCNN モデルは次のように入力を出力へ変換する。

1. 複数の 1 次元畳み込みカーネルを定義し、入力に対してそれぞれ個別に畳み込み演算を行う。幅の異なる畳み込みカーネルは、異なる数の隣接トークン間の局所特徴を捉えられる。
2. すべての出力チャネルに対して max-over-time pooling を行い、その後、すべてのスカラープーリング出力を連結してベクトルにする。
3. 連結したベクトルを全結合層で出力カテゴリへ変換する。過学習を抑えるためにドロップアウトを使える。

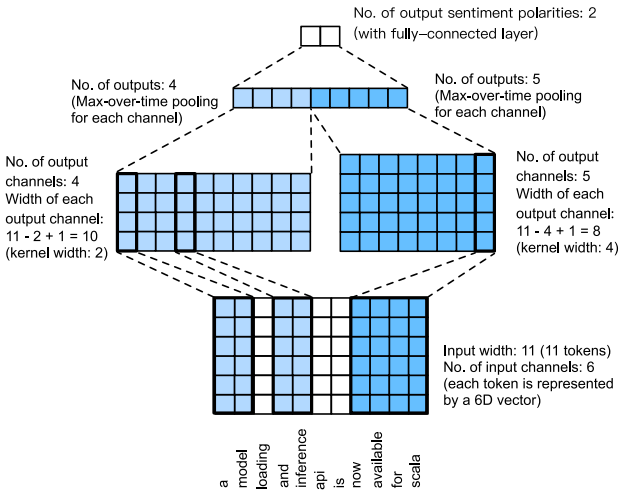


Fig. 19.3.5 textCNN のモデルアーキテクチャ。

Fig. 19.3.5 は、具体例を用いて textCNN のモデルアーキテクチャを示している。

入力は 11 個のトークンからなる文で、各トークンは 6 次元ベクトルで表されている。したがって、幅 11 の 6 チャネル入力がある。幅が 2 と 4 の 2 つの 1 次元畳み込みカーネルを定義し、それぞれ 4 チャネルと 5 チャネルの出力を持たせる。それらは、幅 $11-2+1=10$ の 4 つの出力チャネルと、幅 $11-4+1=8$ の 5 つの出力チャネルを生成する。これら 9 チャネルは幅が異なるが、max-over-time pooling により連結された 9 次元ベクトルが得られ、最終的に 2 次元出力ベクトルへ変換されて二値感情予測を行う。

モデルの定義

以下のクラスで textCNN モデルを実装する。Section 19.2 の双方向 RNN モデルと比べると、再帰層を畳み込み層に置き換えるだけでなく、2 つの埋め込み層も使う。1 つは学習可能な重みを持ち、もう 1 つは固定重みを持つ。

```
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embed_size, kernel_sizes, num_channels,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # 学習しない埋め込み層
        self.constant_embedding = nn.Embedding(vocab_size, embed_size)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Linear(sum(num_channels), 2)
        # max-over-time pooling 層にはパラメータがないので、このインスタンスは
        # 共有できる
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.relu = nn.ReLU()
        # 複数の 1 次元畳み込み層を作成する
        self.convs = nn.ModuleList()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.append(nn.Conv1d(2 * embed_size, c, k))

    def forward(self, inputs):
        # 形状 (batch size, no.
        # of tokens, token vector dimension) の 2 つの埋め込み層の出力をベクト
ル方向に連結する
        embeddings = torch.cat((
            self.embedding(inputs), self.constant_embedding(inputs)), dim=2)
        # 1 次元畳み込み層の入力形式に合わせて、
        # テンソルを並べ替え、2 次元目にチャネルを格納する
        embeddings = embeddings.permute(0, 2, 1)
        # 各 1 次元畳み込み層について、max-over-time
        # pooling の後、形状 (batch size, no. of channels, 1) のテンソルが
        # 得られる。最後の次元を削除し、チャネル方向に連結する
        encoding = torch.cat([
            torch.squeeze(self.relu(self.pool(conv(embeddings))), dim=-1)
            for conv in self.convs], dim=1)
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

```
class TextCNN(nn.Block):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, vocab_size, embed_size, kernel_sizes, num_channels,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # 学習しない埋め込み層
        self.constant_embedding = nn.Embedding(vocab_size, embed_size)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Dense(2)
        # max-over-time pooling 層にはパラメータがないので、このインスタンスは
        # 共有できる
        self.pool = nn.GlobalMaxPool1D()
        # 複数の 1 次元畳み込み層を作成する
        self.convs = nn.Sequential()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.add(nn.Conv1D(c, k, activation='relu'))

    def forward(self, inputs):
        # 形状 (batch size, no.
        # of tokens, token vector dimension) の 2 つの埋め込み層の出力をベクト
ル方向に連結する
        embeddings = np.concatenate((
            self.embedding(inputs), self.constant_embedding(inputs)), axis=2)
        # 1 次元畳み込み層の入力形式に合わせて、
        # テンソルを並べ替え、2 次元目にチャネルを格納する
        embeddings = embeddings.transpose(0, 2, 1)
        # 各 1 次元畳み込み層について、max-over-time
        # pooling の後、形状 (batch size, no. of channels, 1) のテンソルが
        # 得られる。最後の次元を削除し、チャネル方向に連結する
        encoding = np.concatenate([
            np.squeeze(self.pool(conv(embeddings)), axis=-1)
            for conv in self.convs], axis=1)
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

```
d2l.predict_sentiment(net, vocab, 'this movie is so bad')
```

```
d2l.predict_sentiment(net, vocab, 'this movie is so bad')
```

textCNN インスタンスを作成しよう。これは、カーネル幅が 3、4、5 で、すべて 100 個の出力チャネルを持つ 3 つの畳み込み層を備えている。

```
embed_size, kernel_sizes, nums_channels = 100, [3, 4, 5], [100, 100, 100]
devices = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, nums_channels)

def init_weights(module):
    if type(module) in (nn.Linear, nn.Conv1d):
        nn.init.xavier_uniform_(module.weight)

net.apply(init_weights);
```

```
embed_size, kernel_sizes, nums_channels = 100, [3, 4, 5], [100, 100, 100]
devices = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, nums_channels)
net.initialize(init.Xavier(), ctx=devices)
```

```
[07:18:01] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
[07:18:01] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

事前学習済み単語ベクトルの読み込み

Section 19.2 と同様に、事前学習済みの 100 次元 GloVe 埋め込みを初期化済みのトークン表現として読み込みる。これらのトークン表現（埋め込み重み）は、embedding では学習され、constant_embedding では固定される。

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.requires_grad = False
```

```
Downloading ../data/glove.6B.100d.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/glove.6B.100d.zip...
```

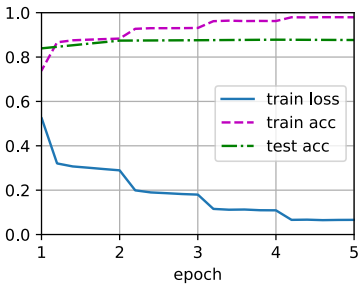
```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.set_data(embeds)
net.constant_embedding.weight.set_data(embeds)
net.constant_embedding.collect_params().setattr('grad_req', 'null')
```

モデルの学習と評価

これで、感情分析のために textCNN モデルを学習できる。

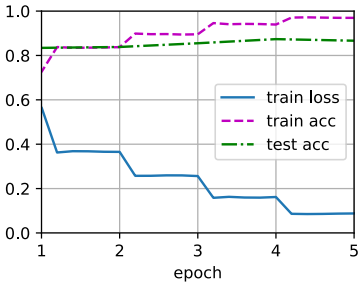
```
lr, num_epochs = 0.001, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.066, train acc 0.979, test acc 0.877
5609.3 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```




```
lr, num_epochs = 0.001, 5
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.088, train acc 0.969, test acc 0.867
1430.3 examples/sec on [gpu(0), gpu(1)]
```



以下では、学習済みモデルを使って 2 つの簡単な文の感情を予測する。

```
d2l.predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
d2l.predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

19.3.4 まとめ

- 1 次元 CNN は、テキスト中の *n*-gram のような局所特徴を処理できる。

- 複数入力チャネルを持つ 1 次元相互相関は、単一入力チャネルの 2 次元相互相関と等価である。
- max-over-time pooling では、チャネルごとに時刻数が異なっていてもよい。
- textCNN モデルは、1 次元畳み込み層と max-over-time pooling 層を用いて、個々のトークン表現を下流アプリケーションの出力へ変換する。

19.3.5 演習

1. ハイパーパラメータを調整し、Section 19.2 とこの節の 2 つのアーキテクチャを、分類精度や計算効率などで比較せよ。
2. Section 19.2 の演習で紹介した手法を用いて、モデルの分類精度をさらに改善できるか。
3. 入力表現に位置エンコーディングを追加せよ。分類精度は向上するか。

19.4 自然言語推論とデータセット

Section 19.1 では、感情分析の問題について議論した。このタスクは、単一のテキスト系列を、感情の極性の集合のような事前に定義されたカテゴリに分類することを目的としている。しかし、ある文が別の文から推論できるかどうかを判断する必要がある場合や、意味的に等価な文を識別して冗長性を取り除く必要がある場合には、1 つのテキスト系列を分類する方法を知っているだけでは不十分である。その代わりに、テキスト系列のペアについて推論できる必要がある。

19.4.1 自然言語推論

自然言語推論は、どちらもテキスト系列である 仮説が 前提から推論できるかどうかを調べる。言い換えると、自然言語推論はテキスト系列のペア間の論理的関係を決定する。このような関係は通常、次の 3 種類に分類される。

- 含意: 仮説は前提から推論できる。
- 矛盾: 仮説の否定は前提から推論できる。
- 中立: それ以外のすべての場合。

自然言語推論は、テキスト含意認識タスクとしても知られている。たとえば、次のペアは、仮説中の“showing affection”が前提中の“hugging one another”から推論できるため、含意とラベル付けされる。

前提: Two women are hugging each other.

仮説: Two women are showing affection.

次は、“running the coding example”が“sleeping”ではなく“not sleeping”を示しているため、矛盾の例である。

前提: A man is running the coding example from Dive into Deep Learning.

仮説: The man is sleeping.

3 つ目の例は、“are performing for us”という事実からは“famous”も“not famous”も推論できないため、中立の関係を示している。

前提: The musicians are performing for us.

仮説: The musicians are famous.

自然言語推論は、自然言語を理解するうえで中心的な話題でした。情報検索からオープンドメイン質問応答に至るまで、幅広い応用がある。この問題を調べるために、まずは広く使われている自然言語推論のベンチマークデータセットを調べる。

19.4.2 Stanford Natural Language Inference (SNLI) データセット

Stanford Natural Language Inference (SNLI) Corpus は、500000 件を超えるラベル付き英語文ペアの集合です (Bowman *et al.*, 2015)。抽出した SNLI データセットを `../data/snli_1.0` にダウンロードして保存する。

```
from d2l import torch as d2l
import torch
from torch import nn
import os
import re

#@save
d2l.DATA_HUB['SNLI'] = (
    'https://nlp.stanford.edu/projects/snli/snli_1.0.zip',
    '9fcde07509c7e87ec61c640c1b2753d9041758e4')

data_dir = d2l.download_extract('SNLI')
```

```
Downloading ../data/snli_1.0.zip from https://nlp.stanford.edu/projects/snli/
↪snli_1.0.zip...
```

```
from d2l import mxnet as d2l
from mxnet import gluon, np, npx
import os
import re

npx.set_np()

#@save
d2l.DATA_HUB['SNLI'] = (
    'https://nlp.stanford.edu/projects/snli/snli_1.0.zip',
    '9fcde07509c7e87ec61c640c1b2753d9041758e4')
```

(continues on next page)

(continued from previous page)

```
data_dir = d2l.download_extract('SNLI')
```

```
#@save
def read_snli(data_dir, is_train):
    """Read the SNLI dataset into premises, hypotheses, and labels."""
    def extract_text(s):
        # Remove information that will not be used by us
        s = re.sub('\\(', '', s)
        s = re.sub('\\)', '', s)
        # Substitute two or more consecutive whitespace with space
        s = re.sub('\\s{2,}', ' ', s)
        return s.strip()
    label_set = {'entailment': 0, 'contradiction': 1, 'neutral': 2}
    file_name = os.path.join(data_dir, 'snli_1.0_train.txt'
                             if is_train else 'snli_1.0_test.txt')
    with open(file_name, 'r') as f:
        rows = [row.split('\t') for row in f.readlines()[1:]]
    premises = [extract_text(row[1]) for row in rows if row[0] in label_set]
    hypotheses = [extract_text(row[2]) for row in rows if row[0] in label_set]
    labels = [label_set[row[0]] for row in rows if row[0] in label_set]
    return premises, hypotheses, labels
```

```
#@save
def read_snli(data_dir, is_train):
    """Read the SNLI dataset into premises, hypotheses, and labels."""
    def extract_text(s):
        # Remove information that will not be used by us
        s = re.sub('\\(', '', s)
        s = re.sub('\\)', '', s)
        # Substitute two or more consecutive whitespace with space
        s = re.sub('\\s{2,}', ' ', s)
        return s.strip()
    label_set = {'entailment': 0, 'contradiction': 1, 'neutral': 2}
    file_name = os.path.join(data_dir, 'snli_1.0_train.txt'
                             if is_train else 'snli_1.0_test.txt')
    with open(file_name, 'r') as f:
        rows = [row.split('\t') for row in f.readlines()[1:]]
    premises = [extract_text(row[1]) for row in rows if row[0] in label_set]
    hypotheses = [extract_text(row[2]) for row in rows if row[0] in label_set]
    labels = [label_set[row[0]] for row in rows if row[0] in label_set]
    return premises, hypotheses, labels
```

データセットの読み込み

元の SNLI データセットには、実験で本当に必要なものよりもはるかに豊富な情報が含まれている。そこで、データセットの一部だけを抽出する `read_snli` 関数を定義し、前提、仮説、およびそれらのラベルのリストを返すようにする。

```
#@save
def read_snli(data_dir, is_train):
    """Read the SNLI dataset into premises, hypotheses, and labels."""
    def extract_text(s):
        # Remove information that will not be used by us
        s = re.sub('\\(', '', s)
        s = re.sub('\\)', '', s)
        # Substitute two or more consecutive whitespace with space
        s = re.sub('\\s{2,}', ' ', s)
        return s.strip()
    label_set = {'entailment': 0, 'contradiction': 1, 'neutral': 2}
    file_name = os.path.join(data_dir, 'snli_1.0_train.txt'
                             if is_train else 'snli_1.0_test.txt')
    with open(file_name, 'r') as f:
        rows = [row.split('\t') for row in f.readlines()[1:]]
    premises = [extract_text(row[1]) for row in rows if row[0] in label_set]
    hypotheses = [extract_text(row[2]) for row in rows if row[0] in label_set]
    labels = [label_set[row[0]] for row in rows if row[0] in label_set]
    return premises, hypotheses, labels
```


```
#@save
def read_snli(data_dir, is_train):
    """Read the SNLI dataset into premises, hypotheses, and labels."""
    def extract_text(s):
        # Remove information that will not be used by us
        s = re.sub('\\(', '', s)
        s = re.sub('\\)', '', s)
        # Substitute two or more consecutive whitespace with space
        s = re.sub('\\s{2,}', ' ', s)
        return s.strip()
    label_set = {'entailment': 0, 'contradiction': 1, 'neutral': 2}
    file_name = os.path.join(data_dir, 'snli_1.0_train.txt'
                             if is_train else 'snli_1.0_test.txt')
    with open(file_name, 'r') as f:
        rows = [row.split('\t') for row in f.readlines()[1:]]
    premises = [extract_text(row[1]) for row in rows if row[0] in label_set]
    hypotheses = [extract_text(row[2]) for row in rows if row[0] in label_set]
    labels = [label_set[row[0]] for row in rows if row[0] in label_set]
    return premises, hypotheses, labels
```


```
train_data = read_snli(data_dir, is_train=True)
for x0, x1, y in zip(train_data[0][:3], train_data[1][:3], train_data[2][:3]):
    print('premise:', x0)
    print('hypothesis:', x1)
    print('label:', y)
```


```
train_data = read_snli(data_dir, is_train=True)
for x0, x1, y in zip(train_data[0][:3], train_data[1][:3], train_data[2][:3]):
    print('premise:', x0)
    print('hypothesis:', x1)
    print('label:', y)
```


では、前提と仮説の最初の 3 組を表示し、それらのラベルも表示してみまし

ょう（“0”、“1”、“2”はそれぞれ“entailment”、“contradiction”、“neutral”に対応する）。

```
train_data = read_snli(data_dir, is_train=True)
for x0, x1, y in zip(train_data[0][:3], train_data[1][:3], train_data[2][:3]):
    print('premise:', x0)
    print('hypothesis:', x1)
    print('label:', y)
```

```
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is training his horse for a competition .
label: 2
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is at a diner , ordering an omelette .
label: 1
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is outdoors , on a horse .
label: 0
```


```
train_data = read_snli(data_dir, is_train=True)
for x0, x1, y in zip(train_data[0][:3], train_data[1][:3], train_data[2][:3]):
    print('premise:', x0)
    print('hypothesis:', x1)
    print('label:', y)
```

```
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is training his horse for a competition .
label: 2
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is at a diner , ordering an omelette .
label: 1
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is outdoors , on a horse .
label: 0
```


```
test_data = read_snli(data_dir, is_train=False)
for data in [train_data, test_data]:
    print([[row for row in data[2]].count(i) for i in range(3)])
```


```
test_data = read_snli(data_dir, is_train=False)
for data in [train_data, test_data]:
    print([[row for row in data[2]].count(i) for i in range(3)])
```


訓練セットには約 550000 組のペアがあり、テストセットには約 10000 組のペアがある。以下は、訓練セットとテストセットの両方で、3 つのラベル“entailment”、“contradiction”、“neutral”がバランスよく分布していることを示している。

```
test_data = read_snli(data_dir, is_train=False)
for data in [train_data, test_data]:
    print([[row for row in data[2]].count(i) for i in range(3)])
```

```
[183416, 183187, 182764]
[3368, 3237, 3219]
```


```
test_data = read_snli(data_dir, is_train=False)
for data in [train_data, test_data]:
    print([[row for row in data[2]].count(i) for i in range(3)])
```

```
[183416, 183187, 182764]
[3368, 3237, 3219]
```


```
train_iter, test_iter, vocab = load_data_snli(128, 50)
len(vocab)
```


```
train_iter, test_iter, vocab = load_data_snli(128, 50)
len(vocab)
```


データセットを読み込むためのクラスの定義

以下では、Gluon の Dataset クラスを継承して SNLI データセットを読み込むためのクラスを定義する。クラスのコンストラクタにある引数 num_steps はテキスト系列の長さを指定し、各ミニバッチの系列が同じ形状になるようにする。言い換えると、長い系列では最初の num_steps 個を超えるトークンは切り捨てられ、短い系列には長さが num_steps になるまで特別なトークン“<pad>”が追加される。__getitem__ 関数を実装することで、インデックス idx を使って前提、仮説、ラベルに任意にアクセスできる。

```
#@save
class SNLIDataset(torch.utils.data.Dataset):
    """A customized dataset to load the SNLI dataset."""
    def __init__(self, dataset, num_steps, vocab=None):
        self.num_steps = num_steps
        all_premise_tokens = d2l.tokenize(dataset[0])
        all_hypothesis_tokens = d2l.tokenize(dataset[1])
        if vocab is None:
            self.vocab = d2l.Vocab(all_premise_tokens + all_hypothesis_tokens,
                                   min_freq=5, reserved_tokens=['<pad>'])
        else:
            self.vocab = vocab
        self.premises = self._pad(all_premise_tokens)
        self.hypotheses = self._pad(all_hypothesis_tokens)
        self.labels = torch.tensor(dataset[2])
        print('read ' + str(len(self.premises)) + ' examples')
```

(continues on next page)

(continued from previous page)

```
    def _pad(self, lines):
        return torch.tensor([d2l.truncate_pad(
            self.vocab[line], self.num_steps, self.vocab['<pad>'])
                             for line in lines])

    def __getitem__(self, idx):
        return (self.premises[idx], self.hypotheses[idx]), self.labels[idx]

    def __len__(self):
        return len(self.premises)
```

```
#@save
class SNLIDataset(gluon.data.Dataset):
    """A customized dataset to load the SNLI dataset."""
    def __init__(self, dataset, num_steps, vocab=None):
        self.num_steps = num_steps
        all_premise_tokens = d2l.tokenize(dataset[0])
        all_hypothesis_tokens = d2l.tokenize(dataset[1])
        if vocab is None:
            self.vocab = d2l.Vocab(all_premise_tokens + all_hypothesis_tokens,
                                   min_freq=5, reserved_tokens=['<pad>'])
        else:
            self.vocab = vocab
        self.premises = self._pad(all_premise_tokens)
        self.hypotheses = self._pad(all_hypothesis_tokens)
        self.labels = np.array(dataset[2])
        print('read ' + str(len(self.premises)) + ' examples')

    def _pad(self, lines):
        return np.array([d2l.truncate_pad(
            self.vocab[line], self.num_steps, self.vocab['<pad>'])
                         for line in lines])

    def __getitem__(self, idx):
        return (self.premises[idx], self.hypotheses[idx]), self.labels[idx]

    def __len__(self):
        return len(self.premises)
```

```
for X, Y in train_iter:
    print(X[0].shape)
    print(X[1].shape)
    print(Y.shape)
    break
```

```
for X, Y in train_iter:
    print(X[0].shape)
    print(X[1].shape)
    print(Y.shape)
    break
```

まとめて読み込む

これで read_snli 関数と SNLIDataset クラスを呼び出して SNLI データセットをダウンロードし、訓練セットとテストセットの両方について DataLoader インスタンスを返し、さらに訓練セットの語彙も得られる。ここで重要なのは、テストセットに対しても訓練セットから構築した語彙を使わなければならないということである。その結果、テストセットに含まれる新しいトークンは、訓練セットで学習したモデルにとって未知のものになる。

```
#@save
def load_data_snli(batch_size, num_steps=50):
    """Download the SNLI dataset and return data iterators and vocabulary."""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('SNLI')
    train_data = read_snli(data_dir, True)
    test_data = read_snli(data_dir, False)
    train_set = SNLIDataset(train_data, num_steps)
    test_set = SNLIDataset(test_data, num_steps, train_set.vocab)
    train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                             shuffle=True,
                                             num_workers=num_workers)
    test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                            shuffle=False,
                                            num_workers=num_workers)
    return train_iter, test_iter, train_set.vocab
```

```
#@save
def load_data_snli(batch_size, num_steps=50):
    """Download the SNLI dataset and return data iterators and vocabulary."""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('SNLI')
    train_data = read_snli(data_dir, True)
    test_data = read_snli(data_dir, False)
    train_set = SNLIDataset(train_data, num_steps)
    test_set = SNLIDataset(test_data, num_steps, train_set.vocab)
    train_iter = gluon.data.DataLoader(train_set, batch_size, shuffle=True,
                                       num_workers=num_workers)
    test_iter = gluon.data.DataLoader(test_set, batch_size, shuffle=False,
                                      num_workers=num_workers)
    return train_iter, test_iter, train_set.vocab
```

ここではバッチサイズを 128、系列長を 50 に設定し、load_data_snli 関数を呼び出してデータ反復子と語彙を取得する。その後、語彙サイズを表示す。

```
train_iter, test_iter, vocab = load_data_snli(128, 50)
len(vocab)
```

```
read 549367 examples
read 9824 examples
```

```
18678
```

次に、最初のミニバッチの形状を表示す。感情分析とは異なり、2 つの入力 `X[0]` と `X[1]` があり、前提と仮説のペアを表す。

```
for X, Y in train_iter:
    print(X[0].shape)
    print(X[1].shape)
    print(Y.shape)
    break
```

```
torch.Size([128, 50])
torch.Size([128, 50])
torch.Size([128])
```

19.4.3 まとめ

- 自然言語推論は、どちらもテキスト系列である前提から仮説が推論できるかどうかを調べる。
- 自然言語推論における前提と仮説の関係には、含意、矛盾、中立がある。
- Stanford Natural Language Inference (SNLI) Corpus は、自然言語推論の代表的なベンチマークデータセットである。

19.4.4 演習

1. 機械翻訳は長い間、出力翻訳と正解翻訳の表面的な *n*-gram の一致に基づいて評価されてきた。自然言語推論を用いて、機械翻訳結果を評価する指標を設計できるか？
2. 語彙サイズを小さくするには、どのようにハイパーパラメータを変更すればよいだろうか？

19.5 自然言語推論: Attention の利用

自然言語推論タスクと SNLI データセットについては、Section 19.4 で導入した。複雑で深いアーキテクチャに基づく多くのモデルを踏まえ、Parikh *et al.* (2016) は attention 機構を用いて自然言語推論に取り組む方法を提案し、これを「decomposable attention model」と呼んだ。その結果、再帰層や畳み込み層を持たないモデルとなり、はるかに少ないパラメータ数で、当時の SNLI データセットにおける最高性能を達成した。この節では、Fig. 19.5.1 に示すような、自然言語推論のためのこの attention ベースの手法（MLP を用いる）を説明し、実装する。

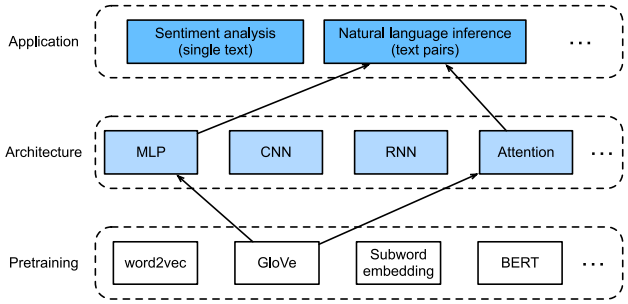


Fig. 19.5.1 この節では、事前学習済み GloVe を、自然言語推論のための attention と MLP に基づくアーキテクチャへ入力する。

19.5.1 モデル

前提文と仮説文におけるトークンの順序を保持するよりも、一方のテキスト系列の各トークンを他方のすべてのトークンに対応付け、その逆も行い、その後でそのような情報を比較・集約して、前提文と仮説文の論理関係を予測すればよいのである。機械翻訳におけるソース文とターゲット文のトークン対応付けと同様に、前提文と仮説文のトークン対応付けは attention 機構によってきれいに実現できる。

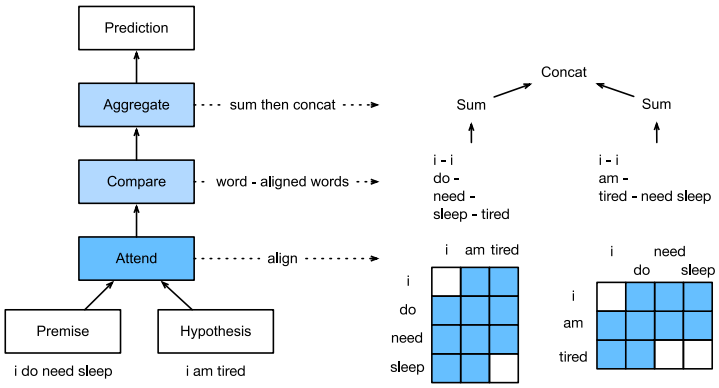


Fig. 19.5.2 Attention 機構を用いた自然言語推論。

Fig. 19.5.2 は、attention 機構を用いた自然言語推論の手法を示している。高レベルでは、これは attending、comparing、aggregating の 3 つのステップを共同で学習する構成である。以下で、それらを順に説明する。

```
from d2l import torch as d2l
import torch
from torch import nn
from torch.nn import functional as F
```

```
from d2l import mxnet as d2l
```

(continues on next page)

(continued from previous page)

```
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn

npx.set_np()
```

```
batch_size, num_steps = 256, 50
train_iter, test_iter, vocab = d2l.load_data_snli(batch_size, num_steps)
```


```
batch_size, num_steps = 256, 50
train_iter, test_iter, vocab = d2l.load_data_snli(batch_size, num_steps)
```


Attending

最初のステップは、一方のテキスト系列の各トークンを、他方の系列の各トークンに対応付けることである。前提文が“i do need sleep”、仮説文が“i am tired”だとしよう。意味的な類似性により、仮説文中の“i”を前提文中の“i”に対応付け、仮説文中の“tired”を前提文中の“sleep”に対応付けたいと考える。同様に、前提文中の“i”を仮説文中の“i”に対応付け、前提文中の“need”と“sleep”を仮説文中の“tired”に対応付けたいと考える。このような対応付けは、重み付き平均を用いた *soft* なものである。理想的には、大きな重みが対応付けたいトークンに割り当てられる。説明を簡単にするため、Fig. 19.5.2 ではそのような対応付けを *hard* な形で示している。

ここでは、attention 機構を用いた soft な対応付けをより詳しく説明する。前提文と仮説文をそれぞれ $\mathbf{A} = (\mathbf{a}_1, \ldots, \mathbf{a}_m)$ および $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ と表す。それぞれのトークン数は m と n であり、$\mathbf{a}_i, \mathbf{b}_j \in \mathbb{R}^d$ $(i = 1, \ldots, m, j = 1, \ldots, n)$ は d 次元の単語ベクトルである。soft な対応付けのために、attention 重み $e_{ij} \in \mathbb{R}$ を次のように計算する。

$$e_{ij} = f(\mathbf{a}_i)^\top f(\mathbf{b}_j), \tag{19.5.1}$$

ここで関数 f は、以下の `mlp` 関数で定義される MLP である。f の出力次元は `mlp` の `num_hiddens` 引数で指定される。

```
def mlp(num_inputs, num_hiddens, flatten):
    net = []
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_inputs, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_hiddens, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    return nn.Sequential(*net)
```

```
def mlp(num_hiddens, flatten):
    net = nn.Sequential()
    net.add(nn.Dropout(0.2))
    net.add(nn.Dense(num_hiddens, activation='relu', flatten=flatten))
    net.add(nn.Dropout(0.2))
    net.add(nn.Dense(num_hiddens, activation='relu', flatten=flatten))
    return net
```

```
predict_snli(net, vocab, ['he', 'is', 'good', '.'], ['he', 'is', 'bad', '.'])
```


```
predict_snli(net, vocab, ['he', 'is', 'good', '.'], ['he', 'is', 'bad', '.'])
```


強調しておくべき点は、(19.5.1) において f は入力として $\mathbf{a}_i$ と $\mathbf{b}_j$ を別々に受け取り、2 つをまとめて入力するわけではないことである。この 分解の工夫により、f の適用回数は mn 回（計算量は二次）ではなく、$m+n$ 回（計算量は線形）で済みる。

(19.5.1) の attention 重みを正規化し、仮説文中のすべてのトークンベクトルの重み付き平均を計算して、前提文中の i 番目のトークンに soft に対応付けられた仮説文の表現を得る。

$$\boldsymbol{\beta}_i = \sum_{j=1}^{n} \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})} \mathbf{b}_j. \tag{19.5.2}$$

同様に、仮説文中の各トークン j に対して、前提文トークンの soft な対応付けを計算する。

$$\boldsymbol{\alpha}_j = \sum_{i=1}^{m} \frac{\exp(e_{ij})}{\sum_{k=1}^{m} \exp(e_{kj})} \mathbf{a}_i. \tag{19.5.3}$$

以下では、入力前提文 `A` に対する仮説文の soft な対応付け（`beta`）と、入力仮説文 `B` に対する前提文の soft な対応付け（`alpha`）を計算する `Attend` クラスを定義する。

```
class Attend(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
        super(Attend, self).__init__(**kwargs)
        self.f = mlp(num_inputs, num_hiddens, flatten=False)

    def forward(self, A, B):
        # Shape of `A`/`B`: (`batch_size`, no. of tokens in sequence A/B,
        # `embed_size`)
        # Shape of `f_A`/`f_B`: (`batch_size`, no. of tokens in sequence A/B,
        # `num_hiddens`)
        f_A = self.f(A)
        f_B = self.f(B)
        # Shape of `e`: (`batch_size`, no. of tokens in sequence A,
        # no. of tokens in sequence B)
        e = torch.bmm(f_A, f_B.permute(0, 2, 1))
```

(continues on next page)

(continued from previous page)

```
        # Shape of `beta`: (`batch_size`, no. of tokens in sequence A,
        # `embed_size`), where sequence B is softly aligned with each token
        # (axis 1 of `beta`) in sequence A
        beta = torch.bmm(F.softmax(e, dim=-1), B)
        # Shape of `alpha`: (`batch_size`, no. of tokens in sequence B,
        # `embed_size`), where sequence A is softly aligned with each token
        # (axis 1 of `alpha`) in sequence B
        alpha = torch.bmm(F.softmax(e.permute(0, 2, 1), dim=-1), A)
        return beta, alpha
```

```
class Attend(nn.Block):
    def __init__(self, num_hiddens, **kwargs):
        super(Attend, self).__init__(**kwargs)
        self.f = mlp(num_hiddens=num_hiddens, flatten=False)

    def forward(self, A, B):
        # Shape of `A`/`B`: (b`atch_size`, no. of tokens in sequence A/B,
        # `embed_size`)
        # Shape of `f_A`/`f_B`: (`batch_size`, no. of tokens in sequence A/B,
        # `num_hiddens`)
        f_A = self.f(A)
        f_B = self.f(B)
        # Shape of `e`: (`batch_size`, no. of tokens in sequence A,
        # no. of tokens in sequence B)
        e = npx.batch_dot(f_A, f_B, transpose_b=True)
        # Shape of `beta`: (`batch_size`, no. of tokens in sequence A,
        # `embed_size`), where sequence B is softly aligned with each token
        # (axis 1 of `beta`) in sequence A
        beta = npx.batch_dot(npx.softmax(e), B)
        # Shape of `alpha`: (`batch_size`, no. of tokens in sequence B,
        # `embed_size`), where sequence A is softly aligned with each token
        # (axis 1 of `alpha`) in sequence B
        alpha = npx.batch_dot(npx.softmax(e.transpose(0, 2, 1)), A)
        return beta, alpha
```

Comparing

次のステップでは、一方の系列のトークンと、そのトークンに soft に対応付けられた他方の系列を比較する。soft な対応付けでは、一方の系列のすべてのトークンが、重みはおそらく異なるものの、他方の系列のあるトークンと比較される。説明を簡単にするため、Fig. 19.5.2 では対応付けられたトークン同士を *hard* な形で組にしている。たとえば、attending ステップによって、前提文中の“need”と“sleep”の両方が仮説文中の“tired”に対応付けられたとすると、“tired–need sleep”の組が比較される。

比較ステップでは、一方の系列のトークンと、他方の系列から対応付けられたトークンの連結（演算子 $[\cdot,\cdot]$）を関数 g（MLP）に入力する。

$$\begin{aligned}\mathbf{v}_{A,i} &= g([\mathbf{a}_i, \boldsymbol{\beta}_i]), i = 1, \ldots, m\\ \mathbf{v}_{B,j} &= g([\mathbf{b}_j, \boldsymbol{\alpha}_j]), j = 1, \ldots, n.\end{aligned} \tag{19.5.4}$$

(19.5.4) において、$\mathbf{v}_{A,i}$ は、前提文中のトークン i と、そのトークン i に soft に対応付けられたすべての仮説文トークンとの比較を表す。一方、$\mathbf{v}_{B,j}$ は、仮説文中のトークン j と、そのトークン j に soft に対応付けられたすべての前提文トークンとの比較を表す。以下の Compare クラスは、この比較ステップを定義する。

```
class Compare(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
        super(Compare, self).__init__(**kwargs)
        self.g = mlp(num_inputs, num_hiddens, flatten=False)

    def forward(self, A, B, beta, alpha):
        V_A = self.g(torch.cat([A, beta], dim=2))
        V_B = self.g(torch.cat([B, alpha], dim=2))
        return V_A, V_B
```

```
class Compare(nn.Block):
    def __init__(self, num_hiddens, **kwargs):
        super(Compare, self).__init__(**kwargs)
        self.g = mlp(num_hiddens=num_hiddens, flatten=False)

    def forward(self, A, B, beta, alpha):
        V_A = self.g(np.concatenate([A, beta], axis=2))
        V_B = self.g(np.concatenate([B, alpha], axis=2))
        return V_A, V_B
```

Aggregating

2 つの比較ベクトル集合 $\mathbf{v}_{A,i}$ $(i = 1, \ldots, m)$ と $\mathbf{v}_{B,j}$ $(j = 1, \ldots, n)$ が得られたら、最後のステップでは、それらの情報を集約して論理関係を推論する。まず、両方の集合をそれぞれ総和する。

$$\mathbf{v}_A = \sum_{i=1}^{m} \mathbf{v}_{A,i}, \quad \mathbf{v}_B = \sum_{j=1}^{n} \mathbf{v}_{B,j}. \tag{19.5.5}$$

次に、両方の要約結果を連結して関数 h（MLP）に入力し、論理関係の分類結果を得る。

$$\hat{\mathbf{y}} = h([\mathbf{v}_A, \mathbf{v}_B]). \tag{19.5.6}$$

集約ステップは、以下の Aggregate クラスで定義される。

```
class Aggregate(nn.Module):
    def __init__(self, num_inputs, num_hiddens, num_outputs, **kwargs):
        super(Aggregate, self).__init__(**kwargs)
        self.h = mlp(num_inputs, num_hiddens, flatten=True)
        self.linear = nn.Linear(num_hiddens, num_outputs)

    def forward(self, V_A, V_B):
        # Sum up both sets of comparison vectors
```

(continues on next page)

(continued from previous page)

```
        V_A = V_A.sum(dim=1)
        V_B = V_B.sum(dim=1)
        # Feed the concatenation of both summarization results into an MLP
        Y_hat = self.linear(self.h(torch.cat([V_A, V_B], dim=1)))
        return Y_hat
```

```
class Aggregate(nn.Block):
    def __init__(self, num_hiddens, num_outputs, **kwargs):
        super(Aggregate, self).__init__(**kwargs)
        self.h = mlp(num_hiddens=num_hiddens, flatten=True)
        self.h.add(nn.Dense(num_outputs))

    def forward(self, V_A, V_B):
        # Sum up both sets of comparison vectors
        V_A = V_A.sum(axis=1)
        V_B = V_B.sum(axis=1)
        # Feed the concatenation of both summarization results into an MLP
        Y_hat = self.h(np.concatenate([V_A, V_B], axis=1))
        return Y_hat
```

全体をまとめる

attending、comparing、aggregating の各ステップを組み合わせることで、これら 3 つのステップを共同で学習する decomposable attention model を定義する。

```
class DecomposableAttention(nn.Module):
    def __init__(self, vocab, embed_size, num_hiddens, num_inputs_attend=100,
                 num_inputs_compare=200, num_inputs_agg=400, **kwargs):
        super(DecomposableAttention, self).__init__(**kwargs)
        self.embedding = nn.Embedding(len(vocab), embed_size)
        self.attend = Attend(num_inputs_attend, num_hiddens)
        self.compare = Compare(num_inputs_compare, num_hiddens)
        # There are 3 possible outputs: entailment, contradiction, and neutral
        self.aggregate = Aggregate(num_inputs_agg, num_hiddens, num_outputs=3)

    def forward(self, X):
        premises, hypotheses = X
        A = self.embedding(premises)
        B = self.embedding(hypotheses)
        beta, alpha = self.attend(A, B)
        V_A, V_B = self.compare(A, B, beta, alpha)
        Y_hat = self.aggregate(V_A, V_B)
        return Y_hat
```

```
class DecomposableAttention(nn.Block):
    def __init__(self, vocab, embed_size, num_hiddens, **kwargs):
        super(DecomposableAttention, self).__init__(**kwargs)
        self.embedding = nn.Embedding(len(vocab), embed_size)
        self.attend = Attend(num_hiddens)
        self.compare = Compare(num_hiddens)
```

(continues on next page)

(continued from previous page)

```
        # There are 3 possible outputs: entailment, contradiction, and neutral
        self.aggregate = Aggregate(num_hiddens, 3)

    def forward(self, X):
        premises, hypotheses = X
        A = self.embedding(premises)
        B = self.embedding(hypotheses)
        beta, alpha = self.attend(A, B)
        V_A, V_B = self.compare(A, B, beta, alpha)
        Y_hat = self.aggregate(V_A, V_B)
        return Y_hat
```

19.5.2 モデルの学習と評価

ここでは、定義した decomposable attention model を SNLI データセットで学習し、評価する。まずデータセットを読み込みる。

データセットの読み込み

Section 19.4 で定義した関数を用いて、SNLI データセットをダウンロードして読み込む。バッチサイズと系列長はそれぞれ 256 と 50 に設定する。

```
batch_size, num_steps = 256, 50
train_iter, test_iter, vocab = d2l.load_data_snli(batch_size, num_steps)
```

```
read 549367 examples
read 9824 examples
```

モデルの作成

入力トークンを表現するために、事前学習済みの 100 次元 GloVe 埋め込みを用いる。したがって、(19.5.1) におけるベクトル $\mathbf{a}_i$ と $\mathbf{b}_j$ の次元を 100 にあらかじめ定める。(19.5.1) における関数 f と、(19.5.4) における関数 g の出力次元は 200 に設定する。その後、モデルインスタンスを作成し、パラメータを初期化し、GloVe 埋め込みを読み込んで入力トークンのベクトルを初期化する。

```
embed_size, num_hiddens, devices = 100, 200, d2l.try_all_gpus()
net = DecomposableAttention(vocab, embed_size, num_hiddens)
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds);
```

```
embed_size, num_hiddens, devices = 100, 200, d2l.try_all_gpus()
net = DecomposableAttention(vocab, embed_size, num_hiddens)
net.initialize(init.Xavier(), ctx=devices)
```

(continues on next page)

(continued from previous page)

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.set_data(embeds)
```

```
[07:30:37] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
[07:30:37] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

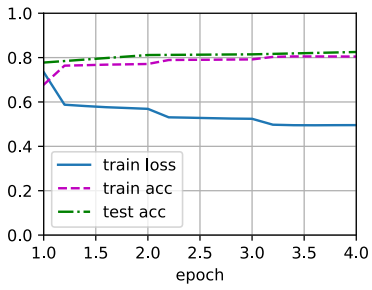
モデルの学習と評価

Section 16.5 にある、テキスト系列や画像のような単一入力を受け取る split_batch 関数とは対照的に、ここでは前提文と仮説文のような複数入力をミニバッチで受け取る split_batch_multi_inputs 関数を定義する。

これで、SNLI データセット上でモデルを学習・評価できる。

```
lr, num_epochs = 0.001, 4
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.496, train acc 0.805, test acc 0.825
15396.1 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```




```
#@save
def split_batch_multi_inputs(X, y, devices):
    """Split multi-input `X` and `y` into multiple devices."""
    X = list(zip(*[gluon.utils.split_and_load(
        feature, devices, even_split=False) for feature in X]))
    return (X, gluon.utils.split_and_load(y, devices, even_split=False))
```

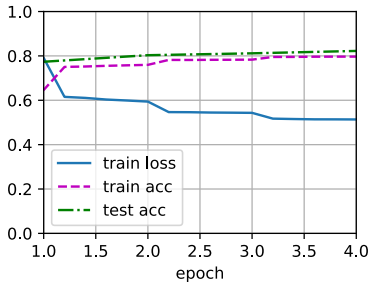
モデルの利用

最後に、前提文と仮説文のペアに対する論理関係を出力する予測関数を定義する。

```
#@save
def predict_snli(net, vocab, premise, hypothesis):
    """Predict the logical relationship between the premise and hypothesis."""
    net.eval()
    premise = torch.tensor(vocab[premise], device=d2l.try_gpu())
    hypothesis = torch.tensor(vocab[hypothesis], device=d2l.try_gpu())
    label = torch.argmax(net([premise.reshape((1, -1)),
                          hypothesis.reshape((1, -1))]), dim=1)
    return 'entailment' if label == 0 else 'contradiction' if label == 1 \
            else 'neutral'
```

```
lr, num_epochs = 0.001, 4
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices,
               split_batch_multi_inputs)
```

```
loss 0.514, train acc 0.797, test acc 0.822
10085.0 examples/sec on [gpu(0), gpu(1)]
```



学習済みモデルを使って、文のペアの自然言語推論結果を得ることができる。

```
predict_snli(net, vocab, ['he', 'is', 'good', '.'], ['he', 'is', 'bad', '.'])
```

```
'contradiction'
```

```
#@save
def predict_snli(net, vocab, premise, hypothesis):
    """Predict the logical relationship between the premise and hypothesis."""
    premise = np.array(vocab[premise], ctx=d2l.try_gpu())
    hypothesis = np.array(vocab[hypothesis], ctx=d2l.try_gpu())
```

(continues on next page)

(continued from previous page)

```
    label = np.argmax(net([premise.reshape((1, -1)),
                           hypothesis.reshape((1, -1))]), axis=1)
    return 'entailment' if label == 0 else 'contradiction' if label == 1 \
            else 'neutral'
```

19.5.3 まとめ

- decomposable attention model は、前提文と仮説文の論理関係を予測するために、attending、comparing、aggregating の 3 ステップから構成される。
- attention 機構を用いると、一方のテキスト系列の各トークンを他方のすべてのトークンに対応付け、その逆も行える。そのような対応付けは重み付き平均を用いた soft なものであり、理想的には大きな重みが対応付けたいトークンに割り当てられる。
- 分解の工夫により、attention 重みを計算する際の計算量は二次ではなく線形となり、より望ましい性質を持つ。
- 事前学習済みの単語ベクトルを、自然言語推論のような下流の自然言語処理タスクの入力表現として利用できる。

19.5.4 演習

1. 他のハイパーパラメータの組み合わせでモデルを学習してみよう。テストセットでより高い精度を得られるか？
2. 自然言語推論における decomposable attention model の主な欠点は何ですか？
3. 任意の文のペアについて、意味的類似度の程度（たとえば 0 から 1 の連続値）を得たいとする。データセットをどのように収集し、ラベル付けすればよいだろうか？attention 機構を用いたモデルを設計できるか？

19.6 系列レベルおよびトークンレベルのアプリケーションのための BERT のファインチューニング

本章の前の節では、RNN、CNN、attention、MLP などに基づく、自然言語処理アプリケーション向けのさまざまなモデルを設計してきた。これらのモデルは、空間的または時間的な制約がある場合には有用であるが、自然言語処理の各タスクごとに個別のモデルを作ることは、実際には不可能である。Section 18.8 では、事前学習モデルである BERT を紹介した。これは、幅広い自然言語処理タスクに対して最小限のアーキテクチャ変更で済みる。一方で、提案当時、BERT はさまざまな自然言語処理タスクで最先端の性能を改善した。他方で、

Section 18.10 で述べたように、元の BERT モデルの 2 つのバージョンはそれぞれ 1 億 1000 万個と 3 億 4000 万個のパラメータを持つ。したがって、十分な計算資源がある場合には、下流の自然言語処理アプリケーション向けに BERT をファインチューニングすることを検討できる。

以下では、自然言語処理アプリケーションの一部を系列レベルとトークンレベルに一般化する。系列レベルでは、BERT によるテキスト入力の表現を出力ラベルへ変換する方法を、単一テキスト分類とテキスト対分類または回帰で紹介する。トークンレベルでは、テキストタグ付けや質問応答といった新しいアプリケーションを簡単に紹介し、BERT がそれらの入力をどのように表現し、出力ラベルへ変換するかを説明する。ファインチューニングでは、異なるアプリケーション間で BERT に必要な「最小限のアーキテクチャ変更」は、追加の全結合層である。下流アプリケーションの教師あり学習では、追加層のパラメータはゼロから学習され、事前学習済み BERT モデル内のすべてのパラメータはファインチューニングされる。

19.6.1 単一テキスト分類

単一テキスト分類は、1 つのテキスト系列を入力として受け取り、その分類結果を出力する。本章で学んだ感情分析のほかに、Corpus of Linguistic Acceptability (CoLA) も単一テキスト分類のデータセットであり、与えられた文が文法的に許容可能かどうかを判定する (Warstadt *et al.*, 2019)。たとえば、“I should study.”は許容されるが、“I should studying.”は許容されない。

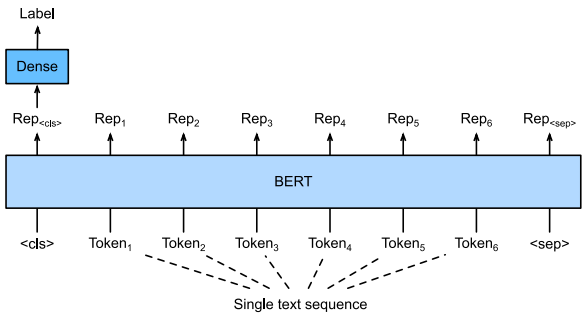


Fig. 19.6.1 感情分析や言語的許容性の判定などの単一テキスト分類アプリケーションのための BERT のファインチューニング。入力の単一テキストは 6 個のトークンを持つと仮定する。

Section 18.8 では BERT の入力表現を説明した。BERT の入力系列は、単一テキストとテキスト対の両方を明確に表現する。ここで特別な分類トークン“<cls>”は系列分類に用いられ、特別な分類トークン“<sep>”は単一テキストの終端を示すか、テキスト対を区切りる。Fig. 19.6.1 に示すように、単一テキスト分類アプリケーションでは、特別な分類トークン“<cls>”の BERT 表現が入力テキスト系列全体の情報を符号化する。入力単一テキストの表現として、これは全結合（dense）層からなる小さな MLP に入力され、すべての離散ラベル値の分布を出力する。

19.6.2 テキスト対分類または回帰

本章では自然言語推論についても見てきた。これは テキスト対分類に属し、テキストのペアを分類するタイプのアプリケーションである。

2 つのテキストを入力として受け取りながら連続値を出力する 意味的テキスト類似度は、人気のある テキスト対回帰タスクである。このタスクは文の意味的類似性を測定する。たとえば、Semantic Textual Similarity Benchmark データセットでは、文のペアの類似度スコアは、0（意味の重なりなし）から 5（意味が等価）までの順序尺度です (Cer *et al.*, 2017)。目的はこれらのスコアを予測することである。Semantic Textual Similarity Benchmark データセットの例には、（文 1、文 2、類似度スコア）として次のものがある。

- “A plane is taking off.”, “An air plane is taking off.”, 5.000;
- “A woman is eating something.”, “A woman is eating meat.”, 3.000;
- “A woman is dancing.”, “A man is talking.”, 0.000.

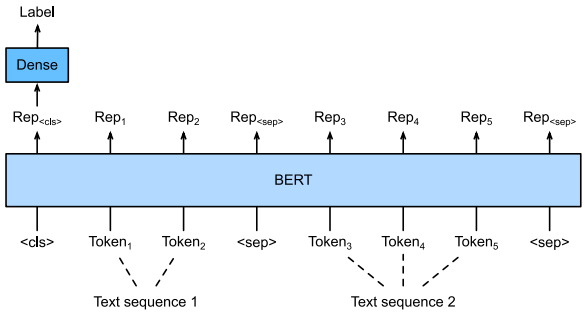


Fig. 19.6.2 自然言語推論や意味的テキスト類似度などのテキスト対分類または回帰アプリケーションのための BERT のファインチューニング。入力のテキスト対は 2 個と 3 個のトークンを持つと仮定する。

Fig. 19.6.1 の単一テキスト分類と比べると、Fig. 19.6.2 におけるテキスト対分類のための BERT のファインチューニングは、入力表現が異なる。意味的テキスト類似度のようなテキスト対回帰タスクでは、連続ラベル値を出力し、平均二乗損失を用いるといった単純な変更を適用できる。これは回帰では一般的である。

19.6.3 テキストタグ付け

次に、テキストタグ付けのようなトークンレベルのタスクを考える。ここでは各トークンにラベルが割り当てられる。テキストタグ付けタスクの中でも、品詞タグ付けは、文中での単語の役割に応じて、各単語に品詞タグ（たとえば形容詞や限定詞）を割り当てる。たとえば、Penn Treebank II のタグセットによれば、文“John Smith ’s car is new”は“NNP (noun, proper singular) NNP POS (possessive ending) NN (noun, singular or mass) VB (verb, base form) JJ (adjective)”とタグ付けされるべきである。

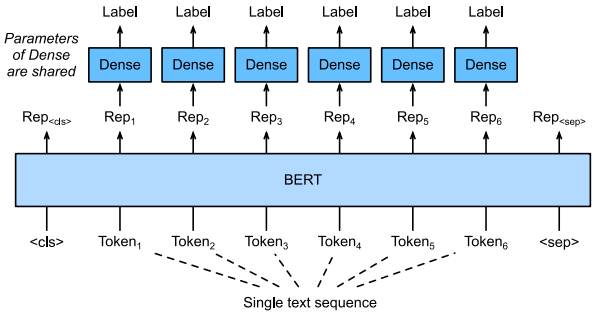


Fig. 19.6.3 品詞タグ付けなどのテキストタグ付けアプリケーションのための BERT のファインチューニング。入力の単一テキストは 6 個のトークンを持つと仮定する。

テキストタグ付けアプリケーションのための BERT のファインチューニングを Fig. 19.6.3 に示す。Fig. 19.6.1 と比べると、唯一の違いは、テキストタグ付けでは、入力テキストの 各トークンの BERT 表現が同じ追加の全結合層に入力され、品詞タグのようなトークンのラベルを出力する点である。

19.6.4 質問応答

別のトークンレベルのアプリケーションとして、質問応答は読解能力を反映する。たとえば、Stanford Question Answering Dataset (SQuAD v1.1) は読解用の文章と質問からなり、各質問への答えは、その質問が関係する文章からのテキスト区間（text span）にすぎない (Rajpurkar *et al.*, 2016)。説明のために、次の文章を考える。“Some experts report that a mask’s efficacy is inconclusive. However, mask makers insist that their products, such as N95 respirator masks, can guard against the virus.”そして質問“Who say that N95 respirator masks can guard against the virus?”を考える。答えは文章中のテキスト区間“mask makers”であるべきである。したがって、SQuAD v1.1 の目的は、質問と文章のペアが与えられたときに、文章中のテキスト区間の開始位置と終了位置を予測することである。

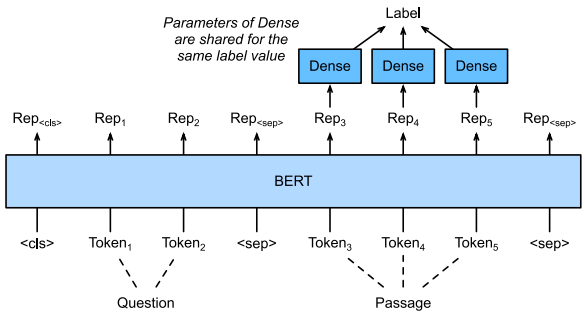


Fig. 19.6.4 質問応答のための BERT のファインチューニング。入力のテキスト対は 2 個と 3 個のトークンを持つと仮定する。

質問応答のために BERT をファインチューニングするには、質問と文章をそれぞれ BERT の入力における第 1 および第 2 のテキスト系列として詰め込みる。テ

キスト区間の開始位置を予測するために、同じ追加の全結合層が、文章中の位置 i にある任意のトークンの BERT 表現をスカラー値 s_i に変換する。文章中のすべてのトークンのこれらのスコアは、さらに softmax 操作によって確率分布へ変換され、文章中の各トークン位置 i に対して、テキスト区間の開始である確率 p_i が割り当てられる。テキスト区間の終了位置の予測も同様であるが、その追加の全結合層のパラメータは開始位置予測用のものとは独立である。終了位置を予測するときには、位置 i にある任意の文章トークンが同じ全結合層によってスカラー値 e_i に変換される。Fig. 19.6.4 は質問応答のための BERT のファインチューニングを示している。

質問応答では、教師あり学習の訓練目的は、正解の開始位置と終了位置の対数尤度を最大化することと同じくらい直接的である。区間を予測するときには、位置 i から位置 j までの有効な区間（$i \leq j$）についてスコア $s_i + e_j$ を計算し、最も高いスコアを持つ区間を出力できる。

19.6.5 まとめ

- BERT は、単一テキスト分類（たとえば感情分析や言語的許容性の判定）、テキスト対分類または回帰（たとえば自然言語推論や意味的テキスト類似度）、テキストタグ付け（たとえば品詞タグ付け）、質問応答などの系列レベルおよびトークンレベルの自然言語処理アプリケーションに対して、最小限のアーキテクチャ変更（追加の全結合層）で対応できる。

- 下流アプリケーションの教師あり学習では、追加層のパラメータはゼロから学習され、事前学習済み BERT モデル内のすべてのパラメータはファインチューニングされる。

19.6.6 演習

1. ニュース記事向けの検索エンジンアルゴリズムを設計しよう。システムがクエリ（たとえば“oil industry during the coronavirus outbreak”）を受け取ったら、そのクエリに最も関連するニュース記事の順位付きリストを返すものとする。大量のニュース記事と多数のクエリがあると仮定する。問題を簡単にするため、各クエリに対して最も関連する記事がラベル付けされていると仮定する。負例サンプリング（Section 18.2.1 を参照）と BERT をアルゴリズム設計にどのように適用できるか？

2. 言語モデルの学習に BERT をどのように活用できるか？

3. 機械翻訳に BERT を活用できるか？

19.7 自然言語推論: BERT のファインチューニング

この章の前半では、SNLI データセット（Section 19.4 で説明）における自然言語推論タスクのために、注意機構に基づくアーキテクチャを設計した（Section 19.5 を参照）。ここでは、このタスクを BERT のファインチューニングによって再び扱いる。Section 19.6 で述べたように、自然言語推論は系列レベルのテキスト対分類問題であり、BERT のファインチューニングには、Fig. 19.7.1 に示すような、追加の MLP ベースのアーキテクチャだけが必要である。

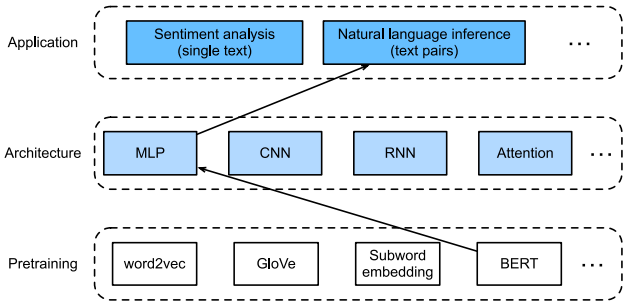


Fig. 19.7.1 この節では、事前学習済み BERT を自然言語推論のための MLP ベースのアーキテクチャに入力する。

この節では、事前学習済みの BERT の小さい版をダウンロードし、SNLI データセット上の自然言語推論のためにそれをファインチューニングする。

```
from d2l import torch as d2l
import json
import multiprocessing
import torch
from torch import nn
import os
```

```
from d2l import mxnet as d2l
import json
import multiprocessing
from mxnet import gluon, np, npx
from mxnet.gluon import nn
import os

npx.set_np()
```

```
devices = d2l.try_all_gpus()
bert, vocab = load_pretrained_model(
```

(continues on next page)

(continued from previous page)

```
    'bert.small', num_hiddens=256, ffn_num_hiddens=512, num_heads=4,
    num_blks=2, dropout=0.1, max_len=512, devices=devices)
```

```
devices = d2l.try_all_gpus()
bert, vocab = load_pretrained_model(
    'bert.small', num_hiddens=256, ffn_num_hiddens=512, num_heads=4,
    num_blks=2, dropout=0.1, max_len=512, devices=devices)
```

19.7.1 事前学習済み BERT の読み込み

WikiText-2 データセット上で BERT を事前学習する方法は、Section 18.9 と Section 18.10 で説明した（なお、元の BERT モデルははるかに大きなコーパスで事前学習されている）。Section 18.10 で述べたように、元の BERT モデルは数億個のパラメータを持っている。以下では、事前学習済み BERT の 2 つの版を用意する。「bert.base」は元の BERT base モデルとほぼ同じ大きさで、ファインチューニングには多くの計算資源を必要とする。一方、「bert.small」はデモをしやすくするための小さい版である。

```
d2l.DATA_HUB['bert.base'] = (d2l.DATA_URL + 'bert.base.torch.zip',
                             '225d66f04cae318b841a13d32af3acc165f253ac')
d2l.DATA_HUB['bert.small'] = (d2l.DATA_URL + 'bert.small.torch.zip',
                              'c72329e68a732bef0452e4b96a1c341c8910f81f')
```

```
d2l.DATA_HUB['bert.base'] = (d2l.DATA_URL + 'bert.base.zip',
                             '7b3820b35da691042e5d34c0971ac3edbd80d3f4')
d2l.DATA_HUB['bert.small'] = (d2l.DATA_URL + 'bert.small.zip',
                              'a4e718a47137ccd1809c9107ab4f5edd317bae2c')
```

どちらの事前学習済み BERT モデルにも、語彙集合を定義する「vocab.json」ファイルと、事前学習済みパラメータを格納した「pretrained.params」ファイルが含まれている。以下の `load_pretrained_model` 関数を実装して、事前学習済み BERT のパラメータを読み込みます。

```
def load_pretrained_model(pretrained_model, num_hiddens, ffn_num_hiddens,
                          num_heads, num_blks, dropout, max_len, devices):
    data_dir = d2l.download_extract(pretrained_model)
    # Define an empty vocabulary to load the predefined vocabulary
    vocab = d2l.Vocab()
    vocab.idx_to_token = json.load(open(os.path.join(data_dir, 'vocab.json')))
    vocab.token_to_idx = {token: idx for idx, token in enumerate(
        vocab.idx_to_token)}
    bert = d2l.BERTModel(
        len(vocab), num_hiddens, ffn_num_hiddens=ffn_num_hiddens, num_heads=4,
        num_blks=2, dropout=0.2, max_len=max_len)
    # Load pretrained BERT parameters
    bert.load_state_dict(torch.load(os.path.join(data_dir,
```

(continues on next page)

(continued from previous page)

```
                                                'pretrained.params')))
    return bert, vocab
```

```
def load_pretrained_model(pretrained_model, num_hiddens, ffn_num_hiddens,
                          num_heads, num_blks, dropout, max_len, devices):
    data_dir = d2l.download_extract(pretrained_model)
    # Define an empty vocabulary to load the predefined vocabulary
    vocab = d2l.Vocab()
    vocab.idx_to_token = json.load(open(os.path.join(data_dir, 'vocab.json')))
    vocab.token_to_idx = {token: idx for idx, token in enumerate(
        vocab.idx_to_token)}
    bert = d2l.BERTModel(len(vocab), num_hiddens, ffn_num_hiddens, num_heads,
                         num_blks, dropout, max_len)
    # Load pretrained BERT parameters
    bert.load_parameters(os.path.join(data_dir, 'pretrained.params'),
                         ctx=devices)
    return bert, vocab
```

多くのマシンでデモを行いやすくするために、この節では事前学習済み BERT の小さい版（「bert.small」）を読み込み、ファインチューニングする。演習では、はるかに大きい「bert.base」をファインチューニングして、テスト精度を大幅に改善する方法を示す。

```
devices = d2l.try_all_gpus()
bert, vocab = load_pretrained_model(
    'bert.small', num_hiddens=256, ffn_num_hiddens=512, num_heads=4,
    num_blks=2, dropout=0.1, max_len=512, devices=devices)
```

```
Downloading ../data/bert.small.torch.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/bert.small.torch.zip...
```

19.7.2 BERT ファインチューニング用データセット

SNLI データセット上の下流タスクである自然言語推論に対して、カスタマイズしたデータセットクラス `SNLIBERTDataset` を定義する。各例では、前提と仮説がテキスト系列のペアを形成し、Fig. 19.6.2 に示すように 1 つの BERT 入力系列にまとめられる。Section 18.8.4 を思い出してほしい。セグメント ID は、BERT 入力系列内で前提と仮説を区別するために使われる。BERT 入力系列の事前定義された最大長（`max_len`）に対して、入力テキスト対のうち長い方の末尾トークンは、`max_len` を満たすまで削除され続ける。BERT のファインチューニングのための SNLI データセット生成を高速化するために、4 つのワーカープロセスを使って訓練例またはテスト例を並列生成する。

```
class SNLIBERTDataset(torch.utils.data.Dataset):
    def __init__(self, dataset, max_len, vocab=None):
        all_premise_hypothesis_tokens = [[
```

(continues on next page)

(continued from previous page)

```
            p_tokens, h_tokens] for p_tokens, h_tokens in zip(
            *[d2l.tokenize([s.lower() for s in sentences])
              for sentences in dataset[:2]])]

        self.labels = torch.tensor(dataset[2])
        self.vocab = vocab
        self.max_len = max_len
        (self.all_token_ids, self.all_segments,
         self.valid_lens) = self._preprocess(all_premise_hypothesis_tokens)
        print('read ' + str(len(self.all_token_ids)) + ' examples')

    def _preprocess(self, all_premise_hypothesis_tokens):
        pool = multiprocessing.Pool(4)  # Use 4 worker processes
        out = pool.map(self._mp_worker, all_premise_hypothesis_tokens)
        all_token_ids = [
            token_ids for token_ids, segments, valid_len in out]
        all_segments = [segments for token_ids, segments, valid_len in out]
        valid_lens = [valid_len for token_ids, segments, valid_len in out]
        return (torch.tensor(all_token_ids, dtype=torch.long),
                torch.tensor(all_segments, dtype=torch.long),
                torch.tensor(valid_lens))

    def _mp_worker(self, premise_hypothesis_tokens):
        p_tokens, h_tokens = premise_hypothesis_tokens
        self._truncate_pair_of_tokens(p_tokens, h_tokens)
        tokens, segments = d2l.get_tokens_and_segments(p_tokens, h_tokens)
        token_ids = self.vocab[tokens] + [self.vocab['<pad>']] \
                              * (self.max_len - len(tokens))
        segments = segments + [0] * (self.max_len - len(segments))
        valid_len = len(tokens)
        return token_ids, segments, valid_len

    def _truncate_pair_of_tokens(self, p_tokens, h_tokens):
        # Reserve slots for '<CLS>', '<SEP>', and '<SEP>' tokens for the BERT
        # input
        while len(p_tokens) + len(h_tokens) > self.max_len - 3:
            if len(p_tokens) > len(h_tokens):
                p_tokens.pop()
            else:
                h_tokens.pop()

    def __getitem__(self, idx):
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx]), self.labels[idx]

    def __len__(self):
        return len(self.all_token_ids)
```

```
class SNLIBERTDataset(gluon.data.Dataset):
    def __init__(self, dataset, max_len, vocab=None):
        all_premise_hypothesis_tokens = [[
            p_tokens, h_tokens] for p_tokens, h_tokens in zip(
            *[d2l.tokenize([s.lower() for s in sentences])
              for sentences in dataset[:2]])]
```

(continues on next page)

(continued from previous page)

```
        self.labels = np.array(dataset[2])
        self.vocab = vocab
        self.max_len = max_len
        (self.all_token_ids, self.all_segments,
         self.valid_lens) = self._preprocess(all_premise_hypothesis_tokens)
        print('read ' + str(len(self.all_token_ids)) + ' examples')

    def _preprocess(self, all_premise_hypothesis_tokens):
        pool = multiprocessing.Pool(4)  # Use 4 worker processes
        out = pool.map(self._mp_worker, all_premise_hypothesis_tokens)
        all_token_ids = [
            token_ids for token_ids, segments, valid_len in out]
        all_segments = [segments for token_ids, segments, valid_len in out]
        valid_lens = [valid_len for token_ids, segments, valid_len in out]
        return (np.array(all_token_ids, dtype='int32'),
                np.array(all_segments, dtype='int32'),
                np.array(valid_lens))

    def _mp_worker(self, premise_hypothesis_tokens):
        p_tokens, h_tokens = premise_hypothesis_tokens
        self._truncate_pair_of_tokens(p_tokens, h_tokens)
        tokens, segments = d2l.get_tokens_and_segments(p_tokens, h_tokens)
        token_ids = self.vocab[tokens] + [self.vocab['<pad>']] \
                              * (self.max_len - len(tokens))
        segments = segments + [0] * (self.max_len - len(segments))
        valid_len = len(tokens)
        return token_ids, segments, valid_len

    def _truncate_pair_of_tokens(self, p_tokens, h_tokens):
        # Reserve slots for '<CLS>', '<SEP>', and '<SEP>' tokens for the BERT
        # input
        while len(p_tokens) + len(h_tokens) > self.max_len - 3:
            if len(p_tokens) > len(h_tokens):
                p_tokens.pop()
            else:
                h_tokens.pop()

    def __getitem__(self, idx):
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx]), self.labels[idx]

    def __len__(self):
        return len(self.all_token_ids)
```

SNLI データセットをダウンロードした後、SNLIBERTDataset クラスをインスタンス化することで、訓練例とテスト例を生成する。これらの例は、自然言語推論の訓練とテストの際にミニバッチで読み込まれる。

```
# Reduce `batch_size` if there is an out of memory error. In the original BERT
# model, `max_len` = 512
batch_size, max_len, num_workers = 512, 128, d2l.get_dataloader_workers()
data_dir = d2l.download_extract('SNLI')
```

(continues on next page)

(continued from previous page)

```
train_set = SNLIBERTDataset(d2l.read_snli(data_dir, True), max_len, vocab)
test_set = SNLIBERTDataset(d2l.read_snli(data_dir, False), max_len, vocab)
train_iter = torch.utils.data.DataLoader(train_set, batch_size, shuffle=True,
                                         num_workers=num_workers)
test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                        num_workers=num_workers)
```

```
read 549367 examples
read 9824 examples
```

```
# Reduce `batch_size` if there is an out of memory error. In the original BERT
# model, `max_len` = 512
batch_size, max_len, num_workers = 512, 128, d2l.get_dataloader_workers()
data_dir = d2l.download_extract('SNLI')
train_set = SNLIBERTDataset(d2l.read_snli(data_dir, True), max_len, vocab)
test_set = SNLIBERTDataset(d2l.read_snli(data_dir, False), max_len, vocab)
train_iter = gluon.data.DataLoader(train_set, batch_size, shuffle=True,
                                   num_workers=num_workers)
test_iter = gluon.data.DataLoader(test_set, batch_size,
                                  num_workers=num_workers)
```

```
Downloading ../data/snli_1.0.zip from https://nlp.stanford.edu/projects/snli/
↪snli_1.0.zip...
read 549367 examples
read 9824 examples
```

19.7.3 BERT のファインチューニング

Fig. 19.6.2 が示すように、自然言語推論のために BERT をファインチューニングするには、2 つの全結合層からなる追加の MLP だけで十分です（以下の `BERTClassifier` クラスの `self.hidden` と `self.output` を参照）。この MLP は、前提と仮説の両方の情報を符号化した特別な“<cls>”トークンの BERT 表現を自然言語推論の 3 つの出力、すなわち含意、矛盾、中立へと変換する。

```
class BERTClassifier(nn.Module):
    def __init__(self, bert):
        super(BERTClassifier, self).__init__()
        self.encoder = bert.encoder
        self.hidden = bert.hidden
        self.output = nn.LazyLinear(3)

    def forward(self, inputs):
        tokens_X, segments_X, valid_lens_x = inputs
        encoded_X = self.encoder(tokens_X, segments_X, valid_lens_x)
        return self.output(self.hidden(encoded_X[:, 0, :]))
```

```
class BERTClassifier(nn.Block):
    def __init__(self, bert):
        super(BERTClassifier, self).__init__()
        self.encoder = bert.encoder
        self.hidden = bert.hidden
        self.output = nn.Dense(3)

    def forward(self, inputs):
        tokens_X, segments_X, valid_lens_x = inputs
        encoded_X = self.encoder(tokens_X, segments_X, valid_lens_x)
        return self.output(self.hidden(encoded_X[:, 0, :]))
```

以下では、事前学習済み BERT モデル `bert` を `BERTClassifier` のインスタンス `net` に入力し、下流アプリケーションに利用する。一般的な BERT ファインチューニングの実装では、追加 MLP の出力層（`net.output`）のパラメータだけがゼロから学習される。事前学習済み BERT エンコーダ（`net.encoder`）と追加 MLP の隠れ層（`net.hidden`）のすべてのパラメータはファインチューニングされる。

```
net = BERTClassifier(bert)
```

```
net = BERTClassifier(bert)
net.output.initialize(ctx=devices)
```

Section 18.8 を思い出してほしい。`MaskLM` クラスと `NextSentencePred` クラスの両方には、それぞれが用いる MLP 内にパラメータがある。これらのパラメータは事前学習済み BERT モデル `bert` の一部であり、したがって `net` のパラメータの一部でもある。しかし、これらのパラメータは事前学習中にマスク言語モデリング損失と次文予測損失を計算するためだけに使われる。これら 2 つの損失関数は下流アプリケーションのファインチューニングには無関係なので、BERT をファインチューニングするときには、`MaskLM` と `NextSentencePred` で用いられる MLP のパラメータは更新されない（staled）。

勾配が古いままのパラメータを許可するために、`d2l.train_batch_ch13` の `step` 関数では `ignore_stale_grad=True` フラグが設定されている。この関数を使って、SNLI の訓練セット（`train_iter`）とテストセット（`test_iter`）を用い、モデル `net` を訓練および評価する。計算資源が限られているため、訓練精度とテスト精度はさらに改善できる。これについては演習で扱いる。

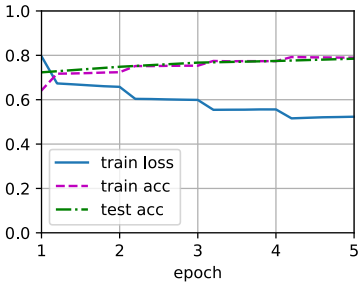
```
lr, num_epochs = 1e-4, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction='none')
net(next(iter(train_iter))[0])
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.523, train acc 0.789, test acc 0.785
```

(continues on next page)

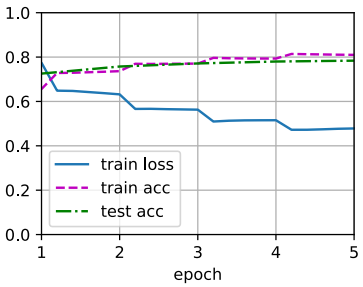
(continued from previous page)

```
10667.1 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```




```
lr, num_epochs = 1e-4, 5
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices,
               d2l.split_batch_multi_inputs)
```

```
loss 0.478, train acc 0.810, test acc 0.784
4400.6 examples/sec on [gpu(0), gpu(1)]
```



19.7.4 まとめ

- 事前学習済み BERT モデルは、SNLI データセット上の自然言語推論のような下流アプリケーションにファインチューニングできる。
- ファインチューニング中、BERT モデルは下流アプリケーションのモデルの一部になる。事前学習の損失にのみ関係するパラメータは、ファインチューニング中には更新されない。

19.7.5 演習

1. 計算資源が許すなら、元の BERT base モデルとほぼ同じ大きさの、はるかに大きい事前学習済み BERT モデルをファインチューニングしよ。`load_pretrained_model`

関数の引数は、'bert.small' を 'bert.base' に置き換え、num_hiddens=256、ffn_num_hiddens=512、num_heads=4、num_blks=2 の値をそれぞれ 768、3072、12、12 に増やしよ。ファインチューニングのエポック数を増やし（必要なら他のハイパーパラメータも調整し）、テスト精度を 0.86 より高くできるか？

2. 長さの比率に応じて 2 つの系列をどのように切り詰めるべきだろうか？このペア切り詰め方法と SNLIBERTDataset クラスで使われている方法を比較しよ。それぞれの長所と短所は何だろうか？

20 強化学習

Pratik Chaudhari (*University of Pennsylvania and Amazon*), **Rasool Fakoor** (*Amazon*), and **Kavosh Asadi** (*Amazon*)

強化学習（RL）は、意思決定を逐次的に行う機械学習システムを構築するための一連の手法である。たとえば、オンライン小売業者から購入した新しい服が入った荷物は、一連の意思決定の結果として玄関先に届く。たとえば、小売業者が自宅に最も近い倉庫で服を見つけ、箱に詰め、陸路または空路で箱を輸送し、都市内のあなたの家まで配達する、といった具合である。荷物の配送過程には多くの変数が影響する。たとえば、服が倉庫に在庫としてあったかどうか、箱の輸送にどれだけ時間がかかったか、毎日の配送トラックが出発する前に荷物があなたの街に到着したかどうか、などである。重要なのは、各段階で私たちがあまり制御できないこれらの変数が、将来の出来事の全体の連鎖に影響を与えるという点である。たとえば、倉庫で箱詰めに遅れが生じた場合、小売業者はタイムリーな配送を確実にするために、陸送ではなく空輸で荷物を送る必要があるかもしれない。強化学習の手法を用いると、逐次意思決定問題の各段階で適切な行動を取り、最終的に何らかの効用、たとえば荷物を時間どおりにあなたへ届けること、を最大化できる。

このような逐次意思決定問題は、ほかにも数多く見られる。たとえば、Go[118]をプレイしているとき、現在の一手が次の一手を決め、相手の手はあなたが制御できない変数である……一連の手が最終的に勝敗を決める。Netflix が今あなたに推薦する映画は、あなたが次に何を見るかを決めるが、その映画を気に入るかどうかは Netflix には分からない。最終的には、一連の映画推薦が Netflix に対する満足度を決定する。強化学習は、今日これらの問題に対する効果的な解決策を開発するために使われている (Mnih *et al.*, 2013, Silver *et al.*, 2016)。強化学習と標準的な深層学習の重要な違いは、標準的な深層学習では、訓練済みモデルがあるテストデータ 1 件に対して行う予測が、将来の別のテストデータの予測に影響しないのに対し、強化学習では将来の時点での意思決定（RL では、意思決定は行動とも呼ばれる）が、過去にどのような意思決定が行われたかに影響されることである。

118



この章では、強化学習の基礎を学び、いくつかの代表的な強化学習手法を実装して実践的な経験を積む。まず、マルコフ決定過程（MDP）と呼ばれる概念を導入し、このような逐次意思決定問題を考えるための枠組みを作る。Value Iteration と呼ばれるアルゴリズムは、MDP における制御されない変数（RL では、これらの制御される変数は環境と呼ばれる）が通常どのように振る舞うかを知っているという仮定のもとで、強化学習問題を解くための最初の手がかりとなる。Value Iteration のより一般的な形である Q-Learning を用いると、環境に

ついて必ずしも完全な知識がなくても適切な行動を取ることができる。次に、専門家の行動を模倣することで、深層ネットワークを強化学習問題にどう用いるかを学ぶ。最後に、未知の環境で行動を選択するために深層ネットワークを用いる強化学習手法を構築する。これらの手法は、今日さまざまな実世界の応用で使われている、より高度な RL アルゴリズムの基盤を成しており、そのいくつかについては本章で触れる。

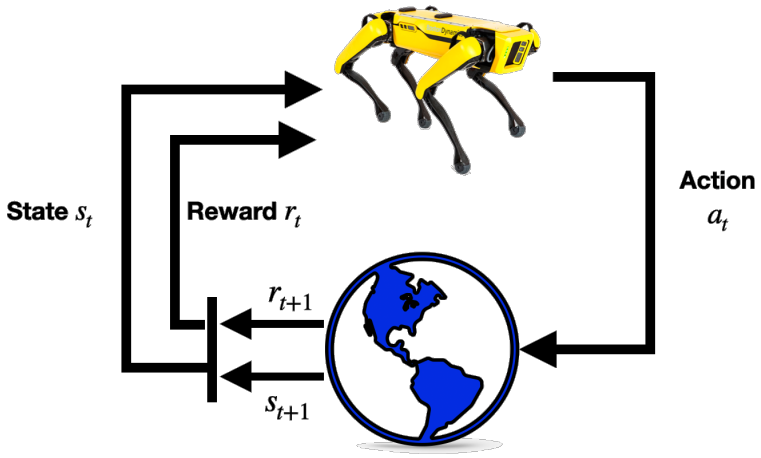


Fig. 20.1 Reinforcement Learning Structure

20.1 マルコフ決定過程（MDP）

この節では、マルコフ決定過程（MDP）を用いて強化学習問題をどのように定式化するかを議論し、MDP のさまざまな構成要素を詳しく説明する。

20.1.1 MDP の定義

マルコフ決定過程（MDP）(Bellman, 1957) は、システムにさまざまな行動が適用されたときに、そのシステムの状態がどのように変化するかを表すモデルである。MDP は、いくつかの異なる量が組み合わさって構成される。

- $\mathcal{S}$ を MDP における状態の集合とする。具体例として、グリッドワールドを移動するロボットについて Fig. 20.1.1 を見よ。この場合、$\mathcal{S}$ は、ロボットが任意の時刻に存在しうる位置の集合に対応する。
- $\mathcal{A}$ を、ロボットが各状態で取りうる行動の集合とする。たとえば、「前進する」「右に曲がる」「左に曲がる」「同じ場所にとどまる」などである。行動によって、ロボットの現在の状態は $\mathcal{S}$ の中の別の状態へ変化しえる。
- ロボットがどのように動くのかを正確には知らず、ある近似までしか分から

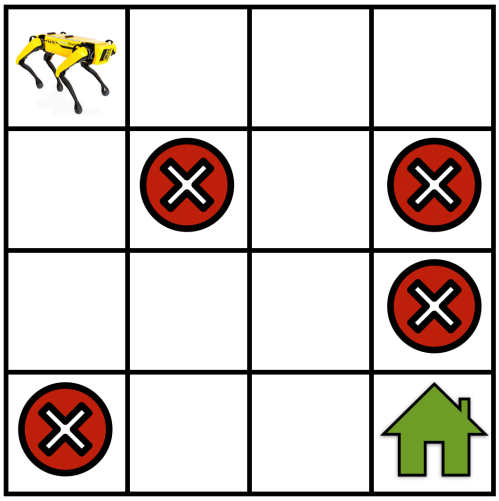


Fig. 20.1.1 A simple gridworld navigation task where the robot not only has to find its way to the goal location (shown as a green house) but also has to avoid trap locations (shown as red cross signs).

ないことがある。強化学習では、この状況を次のようにモデル化する。ロボットが「前進する」という行動を取ったとき、現在の状態にとどまる小さな確率や、「左に曲がる」小さな確率などがあるかもしれない。数学的には、これは「遷移関数」$T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ を定義することに相当し、ロボットが状態 s にいて行動 a を取ったときに状態 s' に到達する条件付き確率を用いて $T(s, a, s') = P(s' \mid s, a)$ と表す。遷移関数は確率分布であるため、すべての $s \in \mathcal{S}$ と $a \in \mathcal{A}$ について $\sum_{s' \in \mathcal{S}} T(s, a, s') = 1$ が成り立つ。すなわち、ロボットは行動を取れば必ず何らかの状態へ移動しなければならない。

- 次に、「どの行動が有用で、どの行動が有用でないか」という概念を、「報酬」$r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ を用いて定義する。ロボットが状態 s で行動 a を取ったとき、報酬 $r(s, a)$ を得るとする。報酬 $r(s, a)$ が大きいなら、それは状態 s で行動 a を取ることが、ロボットの目標、すなわち緑の家に到達することにより有用であることを示す。報酬 $r(s, a)$ が小さいなら、その行動 a はこの目標達成にあまり有用ではない。重要なのは、報酬は目標を念頭に置いて、ユーザ（強化学習アルゴリズムを作成する人）によって設計されるという点である。

20.1.2 リターンと割引率

上のさまざまな構成要素を合わせると、マルコフ決定過程（MDP）

$$\text{MDP} : (\mathcal{S}, \mathcal{A}, T, r). \tag{20.1.1}$$

となる。

では、ロボットが特定の状態 $s_0 \in \mathcal{S}$ から始まり、行動を取り続けて次のような軌道を生じる状況を考えよう。

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots). \tag{20.1.2}$$

各時刻 t において、ロボットは状態 s_t にあり、行動 a_t を取り、その結果として報酬 $r_t = r(s_t, a_t)$ を得る。軌道のリターンとは、そのような軌道に沿ってロボットが得る総報酬のことである。

$$R(\tau) = r_0 + r_1 + r_2 + \cdots. \tag{20.1.3}$$

強化学習の目標は、リターンが最大となる軌道を見つけることである。

ロボットが目標地点に到達することなく、グリッドワールド内を移動し続ける状況を考えてみよ。この場合、軌道における状態と行動の列は無限に長くなりえ、そのような無限長の軌道のリターンは無限大になる。このような軌道に対しても強化学習の定式化を意味のあるものに保つために、割引率 $\gamma < 1$ という概念を導入する。割引されたリターンは次のように書きる。

$$R(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots = \sum_{t=0}^{\infty} \gamma^t r_t. \tag{20.1.4}$$

γ が非常に小さい場合、たとえば $t = 1000$ のような遠い将来にロボットが得る報酬は、γ^{1000} という係数によって大きく割り引かれる。これにより、ロボットは目標を達成する短い軌道、すなわちグリッドワールドの例で緑の家へ向かうこと（Fig. 20.1.1 を参照）を選びやすくなる。割引率が大きい場合、たとえば $\gamma = 0.99$ では、ロボットは探索を行い、その後で目標地点へ向かう最良の軌道を見つけるよう促される。

20.1.3 マルコフ仮定についての考察

新しいロボットを考えてみよう。ここでは、状態 s_t は上と同じく位置であるが、行動 a_t は「前進する」のような抽象的な命令ではなく、ロボットが車輪に加える加速度である。このロボットが状態 s_t で非ゼロの速度を持っているなら、次の位置 s_{t+1} は、過去の位置 s_t、加速度 a_t、さらに時刻 t におけるロボットの速度の関数になる。速度は $s_t - s_{t-1}$ に比例する。これは、次のように書けることを示している。

$$s_{t+1} = \text{some function}(s_t, a_t, s_{t-1}); \tag{20.1.5}$$

この場合の「some function」はニュートンの運動法則に相当する。これは、単に s_t と a_t に依存する遷移関数とはかなり異なる。

マルコフ系とは、次の状態 s_{t+1} が現在の状態 s_t と、現在の状態で取られた行動 a_t のみに依存するようなシステムのことである。マルコフ系では、次の状態は過去にどの行動が取られたか、あるいは過去にロボットがどの状態にいたかには依存しない。たとえば、上で行動が加速度である新しいロボットは、次の位置 s_{t+1} が速度を通じて前の状態 s_{t-1} に依存するため、マルコフ的ではない。システムがマルコフ的であるという性質は制約が強い仮定に見えるかもしれない

が、実際にはそうではない。マルコフ決定過程は、依然として非常に広いクラスの実システムをモデル化できる。たとえば、この新しいロボットについて、状態 s_t を (location, velocity) の組に選べば、次の状態 $(\text{location}_{t+1}, \text{velocity}_{t+1})$ は現在の状態 $(\text{location}_t, \text{velocity}_t)$ と現在の状態での行動 a_t のみに依存するので、システムはマルコフ的になる。

20.1.4 まとめ

強化学習問題は通常、マルコフ決定過程を用いてモデル化される。マルコフ決定過程（MDP）は、4 つの要素 $(\mathcal{S}, \mathcal{A}, T, r)$ の組で定義される。ここで $\mathcal{S}$ は状態空間、$\mathcal{A}$ は行動空間、T は MDP の遷移確率を表す遷移関数、そして r は特定の状態で行動を取ることによって得られる即時報酬である。

20.1.5 演習

1. MountainCar[119] 問題をモデル化する MDP を設計したいとする。
   1. 状態の集合は何になるだろうか？
   2. 行動の集合は何になるだろうか？
   3. 取りうる報酬関数は何だろうか？
2. Pong game[120] のような Atari ゲームのために、どのように MDP を設計するか？

119



120



20.2 値反復

この節では、軌跡の リターンを最大化するために、各状態でロボットが選ぶべき最良の行動をどのように決めるかを議論する。Value Iteration と呼ばれるアルゴリズムを説明し、凍った湖の上を移動するシミュレートされたロボットに対してそれを実装する。

20.2.1 確率的方策

$\pi(a \mid s)$ で表される確率的方策（略して方策）は、状態 $s \in \mathcal{S}$ が与えられたときの行動 $a \in \mathcal{A}$ に関する条件付き分布であり、$\pi(a \mid s) \equiv P(a \mid s)$ である。例として、ロボットに 4 つの行動 $\mathcal{A} = \{$左へ進む, 下へ進む, 右へ進む, 上へ進む$\}$ があるとする。このような行動集合 $\mathcal{A}$ に対する状態 $s \in \mathcal{S}$ での方策はカテゴリ分布であり、4 つの行動の確率は $[0.4, 0.2, 0.1, 0.3]$ かもしれない。別の状態 $s' \in \mathcal{S}$ では、同じ 4 つの行動に対する確率 $\pi(a \mid s')$ は $[0.1, 0.1, 0.2, 0.6]$ かもしれない。任意の状態 s に対して $\sum_a \pi(a \mid s) = 1$ でなければならないことに注意しよ。決定論的方策は確率的方策の特殊な場合であり、$\pi(a \mid s)$ がある特定の 1 つの行動にのみ非ゼロ確率を与えるものである。たとえば、4 つの行動の例では $[1, 0, 0, 0]$ である。

記法を煩雑にしないため、以後はしばしば $\pi(a \mid s)$ の代わりに条件付き分布を $\pi(s)$ と書く。

20.2.2 価値関数

ここで、ロボットが状態 s_0 から始まり、各時刻でまず方策から行動をサンプルし $a_t \sim \pi(s_t)$、その行動を実行して次の状態 s_{t+1} に遷移すると考える。軌跡 $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$ は、中間時刻にどの行動 a_t がサンプルされるかによって変わりえる。このようなすべての軌跡の平均 リターン $R(\tau) = \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)$ を

$$V^\pi(s_0) = E_{a_t \sim \pi(s_t)}\Big[R(\tau)\Big] = E_{a_t \sim \pi(s_t)}\Big[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\Big], \tag{20.2.1}$$

と定義する。

ここで $s_{t+1} \sim P(s_{t+1} \mid s_t, a_t)$ はロボットの次状態であり、$r(s_t, a_t)$ は時刻 t に状態 s_t で行動 a_t を取ることで得られる即時報酬である。これは方策 π に対する「価値関数」と呼ばれる。簡単に言えば、方策 π に対する状態 s_0 の価値、すなわち $V^\pi(s_0)$ は、ロボットが状態 s_0 から始まり、各時刻に方策 π に従って行動したときに得られる、期待 γ-割引 リターンである。

次に、軌跡を 2 つの段階に分ける。(i) 第 1 段階は、行動 a_0 を取ることで $s_0 \to s_1$ となる部分、(ii) 第 2 段階は、その後の軌跡 $\tau' = (s_1, a_1, r_1, \ldots)$ である。強化学習におけるすべてのアルゴリズムの背後にある重要な考え方は、状態 s_0 の価値は、第 1 段階で得られる平均報酬と、ありうる次状態 s_1 にわたって平均した価値関数として書けるということである。これは非常に直感的で、マルコフ仮定から生じる。すなわち、現在の状態からの平均リターンは、次状態からの平均リターンと、次状態へ移ることで得られる平均報酬の和である。数学的には、2 つの段階を次のように書ける。

$$V^\pi(s_0) = r(s_0, a_0) + \gamma\, E_{a_0 \sim \pi(s_0)}\Big[E_{s_1 \sim P(s_1 \mid s_0, a_0)}\Big[V^\pi(s_1)\Big]\Big]. \tag{20.2.2}$$

この分解は非常に強力である。これは動的計画法の原理の基礎であり、すべての強化学習アルゴリズムはこれに基づいている。第 2 段階には 2 つの期待値があることに注意しよ。1 つは確率的方策を用いて第 1 段階で取る行動 a_0 の選択に関する期待、もう 1 つは選ばれた行動から得られる可能な状態 s_1 に関する期待である。マルコフ決定過程（MDP）の遷移確率を用いると、(20.2.2) は次のように書ける。

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s)\Big[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V^\pi(s')\Big]; \text{ for all } s \in \mathcal{S}. \tag{20.2.3}$$

ここで重要なのは、上の等式がすべての状態 $s \in \mathcal{S}$ で成り立つことである。なぜなら、その状態から始まる任意の軌跡を考え、それを 2 つの段階に分解できるからである。

20.2.3 行動価値関数

実装では、しばしば「行動価値」関数と呼ばれる量を保持すると便利である。これは価値関数と密接に関連した量である。これは、s_0 から始まる軌跡の平均リターンであり、ただし第 1 段階の行動を a_0 に固定したものとして定義される。

$$Q^{\pi}(s_0, a_0) = r(s_0, a_0) + E_{a_t \sim \pi(s_t)}\Big[\sum_{t=1}^{\infty} \gamma^t r(s_t, a_t)\Big], \tag{20.2.4}$$

期待値の内側の和が $t = 1, \ldots, \infty$ から始まっているのは、この場合、第 1 段階の報酬が固定されているからである。ここでも軌跡を 2 つに分けて、次のように書ける。

$$Q^{\pi}(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) \sum_{a' \in \mathcal{A}} \pi(a' \mid s')\, Q^{\pi}(s', a'); \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}. \tag{20.2.5}$$

これは行動価値関数に対する (20.2.3) の対応物である。

20.2.4 最適確率的方策

価値関数も行動価値関数も、ロボットが選ぶ方策に依存する。次に、最大の平均 リターンを達成する「最適方策」を考える。

$$\pi^* = \underset{\pi}{\operatorname{argmax}} V^{\pi}(s_0). \tag{20.2.6}$$

ロボットが取りうるすべての確率的方策の中で、最適方策 π^* は、状態 s_0 から始まる軌跡に対して最大の平均割引 リターンを達成する。最適方策の価値関数と行動価値関数を、それぞれ $V^* \equiv V^{\pi^*}$ および $Q^* \equiv Q^{\pi^*}$ と表す。

任意の状態で方策の下で可能な行動が 1 つだけである決定論的方策を考えると、次が得られる。

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \Big[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\, V^*(s')\Big]. \tag{20.2.7}$$

これを覚えるためのよい語呂合わせは、状態 s における最適行動（決定論的方策の場合）は、第 1 段階の報酬 $r(s, a)$ と、第 2 段階でありうるすべての次状態 s' にわたって平均した、次状態 s' から始まる軌跡の平均 リターンの和を最大化するものだ、ということである。

20.2.5 動的計画法の原理

前節の (20.2.2) または (20.2.5) における導出は、それぞれ最適価値関数 V^* または行動価値関数 Q^* を計算するアルゴリズムに変えることができる。次が成り立つ。

$$V^*(s) = \sum_{a \in \mathcal{A}} \pi^*(a \mid s) \Big[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V^*(s')\Big]; \text{ for all } s \in \mathcal{S}. \tag{20.2.8}$$

決定論的な最適方策 π^* では、状態 s で取れる行動は 1 つだけなので、次のようにも書ける。

$$V^*(s) = \text{argmax}_{a \in \mathcal{A}} \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) V^*(s') \right\} \tag{20.2.9}$$

すべての状態 $s \in \mathcal{S}$ に対して成り立つ。この等式は「動的計画法の原理」と呼ばれる (Bellman, 1952, Bellman, 1957)。これは 1950 年代に Richard Bellman によって定式化され、「最適軌跡の残りの部分もまた最適である」と覚えることができる。

20.2.6 値反復

動的計画法の原理を、最適価値関数を求めるアルゴリズム、すなわち値反復に変えることができる。値反復の背後にある重要な考え方は、この等式を、異なる状態 $s \in \mathcal{S}$ における $V^*(s)$ を結びつける制約の集合として捉えることである。まず、すべての状態 $s \in \mathcal{S}$ に対して、価値関数を任意の値 $V_0(s)$ で初期化する。k 回目の反復では、Value Iteration アルゴリズムは価値関数を次のように更新する。

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) V_k(s') \right\}; \text{ for all } s \in \mathcal{S}. \tag{20.2.10}$$

$k \to \infty$ とすると、初期化 V_0 に依らず、Value Iteration アルゴリズムで推定される価値関数は最適価値関数に収束することがわかる。

$$V^*(s) = \lim_{k \to \infty} V_k(s); \text{ for all states } s \in \mathcal{S}. \tag{20.2.11}$$

同じ Value Iteration アルゴリズムは、行動価値関数を用いて次のようにも等価に書ける。

$$Q_{k+1}(s,a) = r(s,a) + \gamma \max_{a' \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s' \mid s,a) Q_k(s',a'); \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}. \tag{20.2.12}$$

この場合、すべての $s \in \mathcal{S}$ と $a \in \mathcal{A}$ に対して $Q_0(s,a)$ を任意の値で初期化する。やはり、すべての $s \in \mathcal{S}$ と $a \in \mathcal{A}$ に対して $Q^*(s,a) = \lim_{k \to \infty} Q_k(s,a)$ が成り立つ。

20.2.7 方策評価

値反復により、最適な決定論的方策 π^* の最適価値関数、すなわち V^{π^*} を計算できる。同様の反復更新を用いて、他の任意の、確率的である可能性もある方策 π に対応する価値関数を計算することもできる。ここでも、すべての状態 $s \in \mathcal{S}$ に対して $V_0^\pi(s)$ を任意の値で初期化し、k 回目の反復で次の更新を行う。

$$V_{k+1}^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) V_k^\pi(s') \right]; \text{ for all } s \in \mathcal{S}. \tag{20.2.13}$$

このアルゴリズムは方策評価として知られており、与えられた方策に対する価値関数を計算するのに役立つ。ここでも、$k \to \infty$ とすると、初期化 V_0 に依ら

ず、これらの更新は正しい価値関数に収束することがわかる。

$$V^{\pi}(s) = \lim_{k \to \infty} V_k^{\pi}(s); \text{ for all states } s \in \mathcal{S}. \tag{20.2.14}$$

方策 π の行動価値関数 $Q^{\pi}(s,a)$ を計算するアルゴリズムも同様である。

20.2.8 値反復の実装

次に、Open AI Gym[121] の FrozenLake と呼ばれるナビゲーション問題に対して、値反復をどのように実装するかを示す。まず、次のコードに示すように環境を設定する必要がある。

121

```
%matplotlib inline
import numpy as np
import random
from d2l import torch as d2l

seed = 0  # Random number generator seed
gamma = 0.95  # Discount factor
num_iters = 10  # Number of iterations
random.seed(seed)  # Set the random seed to ensure results can be reproduced
np.random.seed(seed)

# Now set up the environment
env_info = d2l.make_env('FrozenLake-v1', seed=seed)
```

FrozenLake 環境では、ロボットは 4×4 のグリッド（これらが状態である）上を移動し、行動は「上」(↑)、「下」(→)、「左」(←)、「右」(→) である。環境にはいくつかの穴（H）のセルと凍結した（F）セル、そしてゴールセル（G）があり、これらはすべてロボットには未知である。問題を簡単にするため、ロボットの行動は確実である、すなわちすべての $s \in \mathcal{S}, a \in \mathcal{A}$ に対して $P(s' \mid s,a) = 1$ と仮定する。ロボットがゴールに到達すると、試行は終了し、行動に関係なく報酬 1 を受け取る。それ以外の任意の状態での報酬は、すべての行動に対して 0 である。ロボットの目的は、与えられた開始位置（S）（これが s_0 である）からゴール位置（G）に到達する方策を学習し、リターンを最大化することである。

次の関数は値反復を実装する。`env_info` には MDP と環境に関する情報が含まれ、`gamma` は割引率である。

```
def value_iteration(env_info, gamma, num_iters):
    env_desc = env_info['desc']  # 2D array shows what each item means
    prob_idx = env_info['trans_prob_idx']
    nextstate_idx = env_info['nextstate_idx']
    reward_idx = env_info['reward_idx']
    num_states = env_info['num_states']
    num_actions = env_info['num_actions']
    mdp = env_info['mdp']

    V  = np.zeros((num_iters + 1, num_states))
    Q  = np.zeros((num_iters + 1, num_states, num_actions))
```

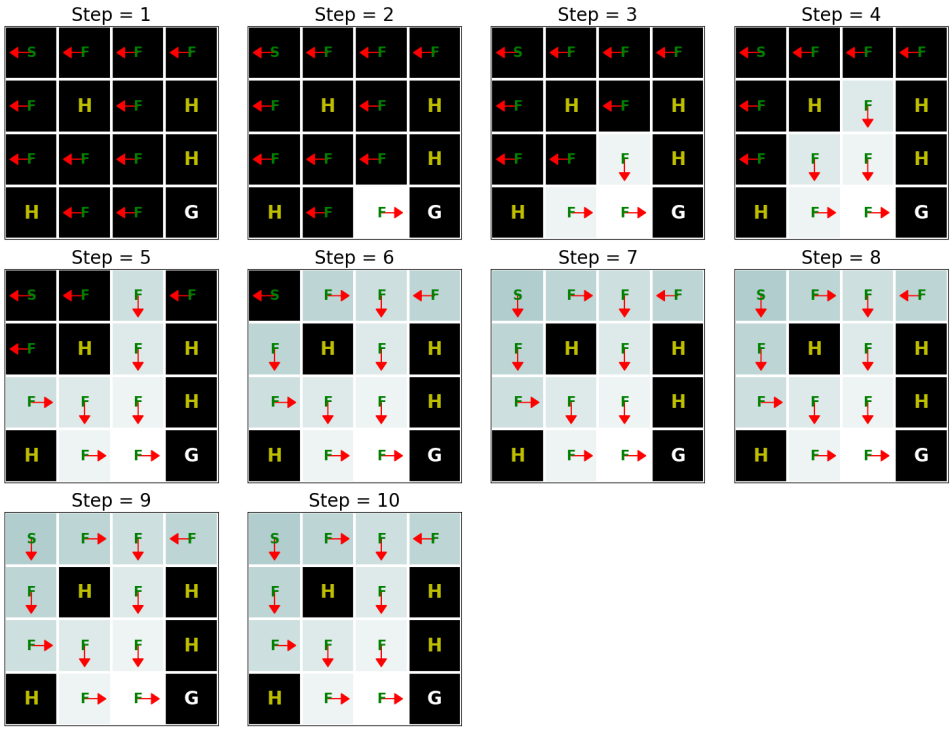
(continues on next page)

(continued from previous page)

```
    pi = np.zeros((num_iters + 1, num_states))

    for k in range(1, num_iters + 1):
        for s in range(num_states):
            for a in range(num_actions):
                # Calculate \sum_{s'} p(s'\mid s,a) [r + \gamma v_k(s')]
                for pxrds in mdp[(s,a)]:
                    # mdp(s,a): [(p1,next1,r1,d1),(p2,next2,r2,d2),..]
                    pr = pxrds[prob_idx]  # p(s'\mid s,a)
                    nextstate = pxrds[nextstate_idx]  # Next state
                    reward = pxrds[reward_idx]  # Reward
                    Q[k,s,a] += pr * (reward + gamma * V[k - 1, nextstate])
            # Record max value and max action
            V[k,s] = np.max(Q[k,s,:])
            pi[k,s] = np.argmax(Q[k,s,:])
    d2l.show_value_function_progress(env_desc, V[:-1], pi[:-1])

value_iteration(env_info=env_info, gamma=gamma, num_iters=num_iters)
```



上の図は、方策（矢印は行動を示す）と価値関数（色の変化は、初期値としての濃い色から最適値としての明るい色へ、価値関数が時間とともにどのように変化するかを示す）を表している。見てわかるように、値反復は 10 回の反復後に最適価値関数を見つけ、H セルでない限り、どの状態から始めてもゴール状態（G）に到達できる。実装のもう 1 つの興味深い点は、最適価値関数を見つけ

るだけでなく、この価値関数に対応する最適方策 π^* も自動的に得られることである。

20.2.9 まとめ

値反復アルゴリズムの主な考え方は、動的計画法の原理を用いて、与えられた状態から得られる最適な平均リターンを求めることである。なお、値反復アルゴリズムを実装するには、マルコフ決定過程（MDP）、たとえば遷移関数や報酬関数を完全に知っている必要がある。

20.2.10 演習

1. グリッドサイズを 8×8 に増やしてみよ。4×4 グリッドと比べて、最適価値関数を見つけるのに何回の反復が必要であるか？
2. 値反復アルゴリズムの計算量はどれくらいか？
3. γ（上のコードの“gamma”）を 0、0.5、1 にしたときに、再度値反復アルゴリズムを実行し、その結果を分析せよ。
4. γ の値は、値反復が収束するまでに要する反復回数にどのように影響するか？$\gamma = 1$ のときはどうなるか？

20.3 Q 学習

前節では、遷移関数や報酬関数など、完全なマルコフ決定過程（MDP）へのアクセスを必要とする価値反復アルゴリズムについて議論した。この節では、MDP を必ずしも知らなくても価値関数を学習できるアルゴリズムである Q 学習 (Watkins and Dayan, 1992) を見ていきる。このアルゴリズムは強化学習の中心的な考え方を体現している。すなわち、ロボットが自分自身のデータを取得できるようにすることである。

20.3.1 Q 学習アルゴリズム

値反復 (page 1279) における行動価値関数の価値反復は、次の更新に対応することを思い出そう。

$$Q_{k+1}(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \max_{a' \in \mathcal{A}} Q_k(s',a'); \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}. \tag{20.3.1}$$

前述したように、このアルゴリズムを実装するには MDP、特に遷移関数 $P(s' \mid s,a)$ を知っている必要がある。Q 学習の鍵となる考え方は、上の式にあるすべての $s' \in \mathcal{S}$ にわたる和を、ロボットが訪れた状態にわたる和で置き換えることである。これにより、遷移関数を知る必要を回避できる。

20.3.2 Q 学習の背後にある最適化問題

ロボットが方策 $\pi_e(a \mid s)$ を使って行動すると想像しよう。前章と同様に、T タイムステップからなる n 本の軌跡 $\{(s_t^i, a_t^i)_{t=0,\ldots,T-1}\}_{i=1,\ldots,n}$ からなるデータセットを収集する。価値反復とは、実際には異なる状態と行動の行動価値 $Q^*(s,a)$ を互いに結び付ける制約の集合であることを思い出しよ。ロボットが π_e を用いて収集したデータを使えば、価値反復の近似版を次のように実装できる。

$$\hat{Q} = \min_Q \underbrace{\frac{1}{nT}\sum_{i=1}^{n}\sum_{t=0}^{T-1}(Q(s_t^i, a_t^i) - r(s_t^i, a_t^i) - \gamma \max_{a'} Q(s_{t+1}^i, a'))^2}_{\stackrel{\text{def}}{=}\ell(Q)}. \tag{20.3.2}$$

まず、この式と上の価値反復との類似点と相違点を見てみよう。もしロボットの方策 π_e が最適方策 π^* と等しく、かつ無限量のデータを収集できるなら、この最適化問題は価値反復の背後にある最適化問題と同一になる。しかし、価値反復では $P(s' \mid s,a)$ を知る必要があるのに対し、この最適化目的関数にはこの項がない。これはごまかしではない。ロボットが状態 s_t^i で方策 π_e を使って行動 a_t^i を選ぶとき、次状態 s_{t+1}^i は遷移関数から引かれたサンプルである。したがって、この最適化目的関数も遷移関数にアクセスしているが、それはロボットが収集したデータという形で暗黙的に与えられているのである。

この最適化問題の変数は、すべての $s \in \mathcal{S}$ と $a \in \mathcal{A}$ に対する $Q(s,a)$ である。勾配降下法を用いて目的関数を最小化できる。データセット中の各組 (s_t^i, a_t^i) について、次のように書ける。

$$\begin{aligned} Q(s_t^i, a_t^i) &\leftarrow Q(s_t^i, a_t^i) - \alpha \nabla_{Q(s_t^i, a_t^i)} \ell(Q) \\ &= (1-\alpha) Q(s_t^i, a_t^i) - \alpha\Big(r(s_t^i, a_t^i) + \gamma \max_{a'} Q(s_{t+1}^i, a')\Big), \end{aligned} \tag{20.3.3}$$

ここで α は学習率である。通常、実際の問題では、ロボットが目標地点に到達すると軌跡は終了する。このような終端状態の価値は 0 である。なぜなら、ロボットはこの状態以降、これ以上行動を取らないからである。このような状態を扱うために、更新式を次のように修正する必要がある。

$$Q(s_t^i, a_t^i) = (1-\alpha) Q(s_t^i, a_t^i) - \alpha\Big(r(s_t^i, a_t^i) + \gamma(1 - \mathbb{1}_{s_{t+1}^i \text{ is terminal}}) \max_{a'} Q(s_{t+1}^i, a')\Big). \tag{20.3.4}$$

ここで $\mathbb{1}_{s_{t+1}^i \text{ is terminal}}$ は指示変数であり、s_{t+1}^i が終端状態なら 1、それ以外なら 0 である。データセットの一部ではない状態-行動の組 (s,a) の値は $-\infty$ に設定される。このアルゴリズムは Q 学習として知られている。

これらの更新の解 $\hat{Q}$、すなわち最適価値関数 Q^* の近似が得られれば、この価値関数に対応する最適な決定論的方策を次のように簡単に得られる。

$$\hat{\pi}(s) = \text{argmax}_a \hat{Q}(s,a). \tag{20.3.5}$$

同じ最適価値関数に対応する決定論的方策が複数存在する場合がある。そのような同値は任意に解いて構わない。なぜなら、それらは同じ価値関数を持つからである。

20.3.3 Q 学習における探索

データ収集にロボットが用いる方策 π_e は、Q 学習がうまく機能するために重要である。結局のところ、私たちは遷移関数 $P(s' \mid s, a)$ による s' に関する期待値を、ロボットが収集したデータで置き換えているのである。もし方策 π_e が状態-行動空間の多様な部分に到達しないなら、推定値 $\hat{Q}$ が最適な Q^* の不十分な近似になることは容易に想像できる。このような状況では、π_e によって訪れた状態だけでなく、すべての状態 $s \in \mathcal{S}$ における Q^* の推定も悪くなることに注意することが重要である。これは、Q 学習の目的関数（あるいは価値反復）が、すべての状態-行動対の価値を結び付ける制約だからである。したがって、データを収集するために適切な方策 π_e を選ぶことが極めて重要である。

この懸念は、$\mathcal{A}$ から一様ランダムに行動をサンプルする完全にランダムな方策 π_e を選ぶことで緩和できる。そのような方策はすべての状態を訪れるが、そうなるまでには大量の軌跡が必要になる。

こうして Q 学習における第 2 の重要な考え方、すなわち探索に至りる。Q 学習の典型的な実装では、現在の Q の推定値と方策 π_e を結び付けて、次のように設定する。

$$\pi_e(a \mid s) = \begin{cases} \text{argmax}_{a'} \hat{Q}(s, a') & \text{with prob. } 1 - \epsilon \\ \text{uniform}(\mathcal{A}) & \text{with prob. } \epsilon, \end{cases} \tag{20.3.6}$$

ここで ϵ は「探索パラメータ」と呼ばれ、ユーザが選ぶ。方策 π_e は探索方策と呼ばれる。この特定の π_e は、現在の推定値 $\hat{Q}$ に基づく最適行動を確率 $1 - \epsilon$ で選び、残りの確率 ϵ でランダムに探索するため、ϵ-greedy 探索方策と呼ばれる。いわゆる softmax 探索方策も使える。

$$\pi_e(a \mid s) = \frac{e^{\hat{Q}(s,a)/T}}{\sum_{a'} e^{\hat{Q}(s,a')/T}}; \tag{20.3.7}$$

ここでハイパーパラメータ T は温度と呼ばれる。ϵ-greedy 方策における大きな ϵ は、softmax 方策における大きな温度 T と同様に機能する。

現在の行動価値関数の推定値 $\hat{Q}$ に依存する探索を選ぶときには、最適化問題を定期的に解き直す必要があることに注意することが重要である。Q 学習の典型的な実装では、π_e を用いて毎回行動を取った後、収集したデータセット中のいくつかの状態-行動対（通常はロボットの直前のタイムステップで収集されたもの）を使って 1 回のミニバッチ更新を行う。

20.3.4 Q 学習の「自己修正」特性

Q 学習中にロボットが収集するデータセットは、時間とともに増えていく。探索方策 π_e も推定値 $\hat{Q}$ も、ロボットがより多くのデータを収集するにつれて変化する。これにより、Q 学習がうまく機能する理由について重要な洞察が得られる。状態 s を考えよう。ある行動 a が現在の推定値 $\hat{Q}(s, a)$ の下で大きな値を持つなら、ϵ-greedy 探索方策も softmax 探索方策も、この行動を選ぶ確率が高くなる。もしこの行動が実際には理想的な行動ではないなら、この行動から生じる将来の状態は低い報酬しか持たない。したがって、次の Q 学習目的関数の更

新では $\hat{Q}(s,a)$ の値が下がり、次にロボットが状態 s を訪れたときにこの行動を選ぶ確率も下がる。悪い行動、たとえば $\hat{Q}(s,a)$ において過大評価されている行動は、ロボットによって探索されるが、Q 学習目的関数の次の更新でその値は修正される。良い行動、たとえば $\hat{Q}(s,a)$ が大きい行動は、ロボットによってより頻繁に探索され、その結果として強化される。この性質を用いると、Q 学習はランダムな方策 π_e から始めたとしても最適方策に収束できることを示せる (Watkins and Dayan, 1992)。

新しいデータを収集するだけでなく、適切な種類のデータを収集する能力こそが強化学習アルゴリズムの中心的な特徴であり、これが教師あり学習との違いである。深層ニューラルネットワークを用いた Q 学習（これは後の DQN 章で見る）は、強化学習の再興を支えた (Mnih *et al.*, 2013)。

20.3.5 Q 学習の実装

ここでは、Open AI Gym[122] の FrozenLake に対して Q 学習を実装する方法を示す。これは値反復 (page 1279) の実験で考えたものと同じ設定である。

122



```
%matplotlib inline
import numpy as np
import random
from d2l import torch as d2l

seed = 0  # Random number generator seed
gamma = 0.95  # Discount factor
num_iters = 256  # Number of iterations
alpha   = 0.9  # Learing rate
epsilon = 0.9  # Epsilon in epsilion gready algorithm
random.seed(seed)  # Set the random seed
np.random.seed(seed)

# Now set up the environment
env_info = d2l.make_env('FrozenLake-v1', seed=seed)
```

FrozenLake 環境では、ロボットは 4×4 のグリッド（これらが状態である）上を移動し、行動として「上」(↑)、「下」(→)、「左」(←)、「右」(→) を取る。環境にはいくつかの穴（H）のセルと凍結した（F）セル、そして目標セル（G）が含まれているが、これらはすべてロボットには未知である。問題を簡単にするため、ロボットの行動は確実である、すなわちすべての $s \in \mathcal{S}, a \in \mathcal{A}$ に対して $P(s' \mid s,a) = 1$ と仮定する。ロボットが目標に到達すると、試行は終了し、行動に関係なく報酬 1 を受け取る。それ以外の状態での報酬は、すべての行動に対して 0 である。ロボットの目的は、与えられた開始位置（S）（これは s_0 である）から目標位置（G）に到達する方策を学習し、収益を最大化することである。

まず、ϵ-greedy 法を次のように実装する。

```
def e_greedy(env, Q, s, epsilon):
    if random.random() < epsilon:
```

(continues on next page)

(continued from previous page)

```
        return env.action_space.sample()

    else:
        return np.argmax(Q[s,:])
```

これで Q 学習を実装する準備が整った。

```
def q_learning(env_info, gamma, num_iters, alpha, epsilon):
    env_desc = env_info['desc']  # 2D array specifying what each grid item␣
↪means
    env = env_info['env']  # 2D array specifying what each grid item means
    num_states = env_info['num_states']
    num_actions = env_info['num_actions']

    Q  = np.zeros((num_states, num_actions))
    V  = np.zeros((num_iters + 1, num_states))
    pi = np.zeros((num_iters + 1, num_states))

    for k in range(1, num_iters + 1):
        # Reset environment
        state, done = env.reset(), False
        while not done:
            # Select an action for a given state and acts in env based on␣
↪selected action
            action = e_greedy(env, Q, state, epsilon)
            next_state, reward, done, _ = env.step(action)

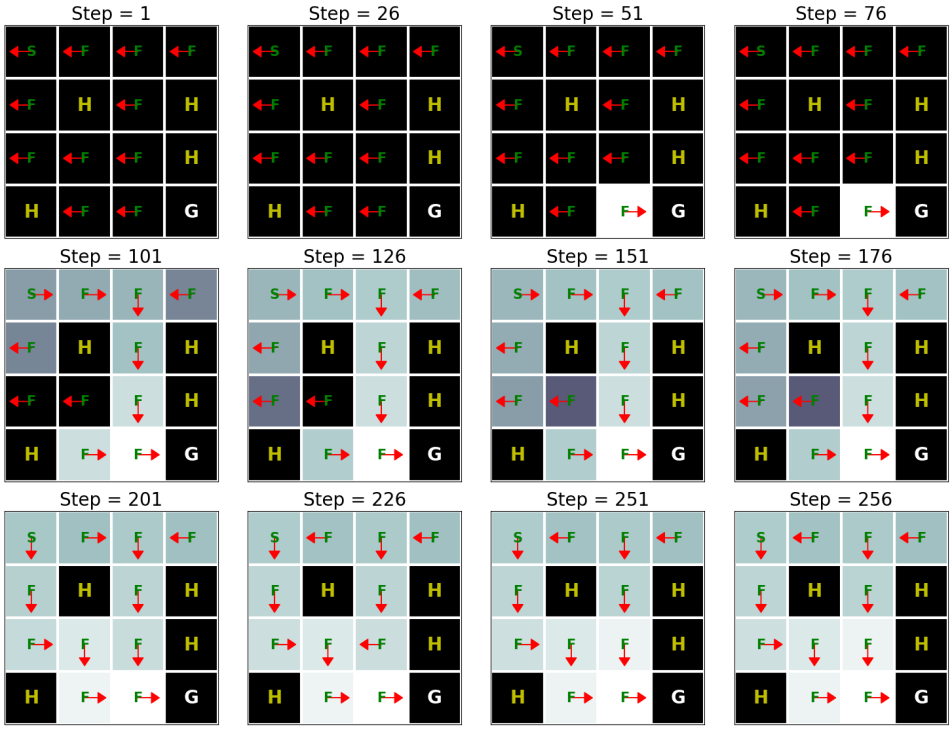
            # Q-update:
            y = reward + gamma * np.max(Q[next_state,:])
            Q[state, action] = Q[state, action] + alpha * (y - Q[state,␣
↪action])

            # Move to the next state
            state = next_state
        # Record max value and max action for visualization purpose only
        for s in range(num_states):
            V[k,s]  = np.max(Q[s,:])
            pi[k,s] = np.argmax(Q[s,:])
    d2l.show_Q_function_progress(env_desc, V[:-1], pi[:-1])

q_learning(env_info=env_info, gamma=gamma, num_iters=num_iters, alpha=alpha,␣
↪epsilon=epsilon)
```

この結果は、Q 学習が約 250 回の反復の後にこの問題の最適解を見つけられることを示している。しかし、この結果を価値反復アルゴリズムの結果（値反復の実装 (page 1283) を参照）と比較すると、価値反復アルゴリズムの方がこの問題の最適解を見つけるのにずっと少ない反復回数で済むことがわかる。これは、価値反復アルゴリズムが完全な MDP にアクセスできるのに対し、Q 学習はそうではないためである。

20.3.6 まとめ



Q 学習は、最も基本的な強化学習アルゴリズムの 1 つである。近年の強化学習の成功、特にビデオゲームをプレイする学習において中心的な役割を果たしてきた (Mnih *et al.*, 2013)。Q 学習の実装には、マルコフ決定過程（MDP）、たとえば遷移関数や報酬関数を完全に知っている必要はない。

20.3.7 演習

1. グリッドサイズを 8×8 に増やしてみよ。4×4 グリッドと比べて、最適価値関数を見つけるのに何回の反復が必要であるか？
2. Q 学習アルゴリズムを、γ（つまり上のコードの“gamma”）を 0、0.5、1 にした場合で再度実行し、その結果を分析せよ。
3. Q 学習アルゴリズムを、ϵ（つまり上のコードの“epsilon”）を 0、0.5、1 にした場合で再度実行し、その結果を分析せよ。

21 ガウス過程

Andrew Gordon Wilson (*New York University and Amazon*)

ガウス過程（GP）は至るところに存在する。あなたはすでに、気づかないうちにガウス過程の多くの例に出会っているはずである。パラメータに対してガウス分布を仮定し、かつパラメータに関して線形な任意のモデルは、ガウス過程である。このクラスには、ランダムウォークや自己回帰過程を含む離散モデルだけでなく、ベイズ線形回帰モデル、多項式、フーリエ級数、ラジアル基底関数、さらには無限個の隠れユニットを持つニューラルネットワークのような連続モデルも含まれる。「すべてはガウス過程の特殊ケースである」という冗談がよく言われる。

ガウス過程を学ぶことが重要なのは、次の 3 つの理由からである。(1) それらはモデリングに対する 関数空間の視点を与え、深層ニューラルネットワークを含むさまざまなモデルクラスの理解をずっと容易にしてくれる。(2) アクティブラーニング、ハイパーパラメータ学習、AutoML、時空間回帰など、最先端性能を示す応用範囲が非常に広いである。(3) ここ数年のアルゴリズムの進歩により、ガウス過程はますますスケーラブルで実用的になり、GPyTorch[123] (Gardner *et al.*, 2018) のようなフレームワークを通じて深層学習と調和するようになった。実際、GP と深層ニューラルネットワークは競合する手法ではなく、非常に補完的であり、組み合わせることで大きな効果を発揮できる。これらのアルゴリズムの進歩はガウス過程にだけ関係するのではなく、深層学習全般に広く役立つ数値計算法の基盤も提供する。

123



この章では、ガウス過程を紹介する。導入ノートブックでは、まずガウス過程とは何か、そしてそれが関数をどのように直接モデル化するのかを直感的に考える。事前分布のノートブックでは、ガウス過程の事前分布をどのように指定するかに焦点を当てる。従来の重み空間でのモデリング手法を関数空間に直接結びつけ、深層ニューラルネットワークを含む機械学習モデルを構築し理解する際の助けとする。次に、ガウス過程の汎化特性を制御する、共分散関数としても知られる人気のある カーネルを紹介する。与えられたカーネルを持つ GP は、関数に対する事前分布を定義する。推論ノートブックでは、予測を行うためにデータを用いて 事後分布を推定する方法を示す。このノートブックには、ガウス過程による予測を行うためのゼロからのコードと、GPyTorch の入門が含まれている。今後のノートブックでは、ガウス過程の背後にある数値計算を紹介する。これはガウス過程をスケールさせるうえで有用であるだけでなく、深層学習のための強力な一般的基盤でもあり、深層学習におけるハイパーパラメータ調整のような高度なユースケースにも役立ちる。例では、ガウス過程をス

ケール可能にする GPyTorch を用い、深層学習機能および PyTorch と密接に統合された形で説明する。

21.1 ガウス過程入門

多くの場合、機械学習はデータからパラメータを推定することに帰着する。これらのパラメータはしばしば多数で、比較的解釈しにくいものです—たとえばニューラルネットワークの重みのように。これに対してガウス過程は、データに適合しうる関数の高レベルな性質について直接推論するための仕組みを提供する。たとえば、これらの関数が急激に変化するのか、周期的なのか、条件付き独立性を含むのか、あるいは平行移動不変なのか、といった感覚を持つことができる。ガウス過程を使うと、データに適合しうる関数値に対するガウス分布を直接指定することで、これらの性質をモデルに容易に組み込める。

まずはいくつかの例から始めて、ガウス過程がどのように働くのかをつかもう。

次のような回帰ターゲット（出力）y を、入力 x によって添字付けされたデータセットを観測したとする。例として、ターゲットは二酸化炭素濃度の変化量であり、入力はそれらが記録された時刻かもしれない。データの特徴は何だろうか？どのくらい速く変化しているように見えるだろうか？データ点は規則的な間隔で収集されているだろうか、それとも入力が欠けているだろうか？欠損している領域をどのように埋めると想像するか、あるいは $x = 25$ までどのように予測するか？

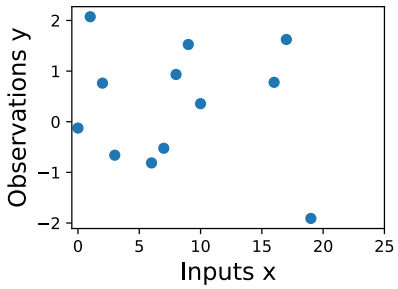


Fig. 21.1.1 Observed data.

ガウス過程でデータに適合させるためには、まず、どのような種類の関数が妥当だと考えるかについての事前分布を指定する。ここでは、ガウス過程からのいくつかのサンプル関数を示す。この事前分布は妥当そうに見えるだろうか？ここで見ているのはデータセットに適合する関数ではなく、入力に対してどのくらい速く変化するかといった、解の高レベルな性質が妥当かどうかである。なお、このノートブックにあるすべての図を再現するコードは、事前分布と推論に関する次のノートブックで示す。

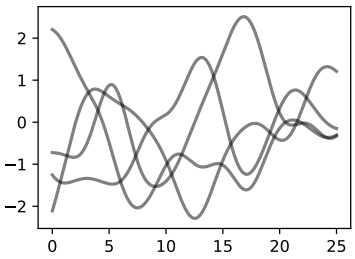


Fig. 21.1.2 Sample prior functions that we may want to represent with our model.

データを条件づけると、この事前分布を用いて、データに適合しうる関数の事後分布を推論できる。ここでは、事後分布からのサンプル関数を示す。

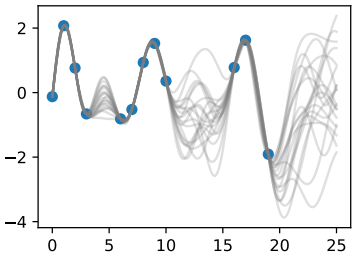


Fig. 21.1.3 Sample posterior functions, once we have observed the data.

これらの関数はそれぞれ、観測データと完全に整合しており、各観測点をぴったり通過している。これらの事後サンプルを使って予測を行うには、事後分布に含まれるあらゆる可能なサンプル関数の値を平均し、下の太い青線の曲線を作ればよいである。なお、この期待値を計算するのに実際に無限個のサンプルを取る必要はない。後で見るように、期待値は閉形式で計算できる。

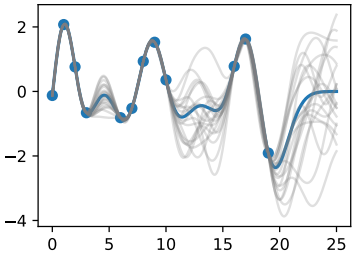


Fig. 21.1.4 Posterior samples, alongside posterior mean, which can be used for point predictions, in blue.

また、予測にどの程度自信を持つべきかを知るために、不確実性の表現も欲しくなる。直感的には、事後サンプル関数のばらつきが大きい場所ほど不確実性も大きいはずである。なぜなら、真の関数が取りうる値の候補がより多いことを示しているからである。この種の不確実性は *epistemic uncertainty* と呼ばれ、

情報不足に伴う *reducible uncertainty* である。データが増えるにつれて、この種の不確実性は消えていく。なぜなら、観測と整合する解が次第に少なくなるからである。事後平均と同様に、事後分散（事後分布におけるこれらの関数のばらつき）も閉形式で計算できる。陰影で示しているのは、平均の両側に事後標準偏差の 2 倍を取ったもので、任意の入力 x に対して真の関数値を 95% の確率で含む *credible interval* を作っている。

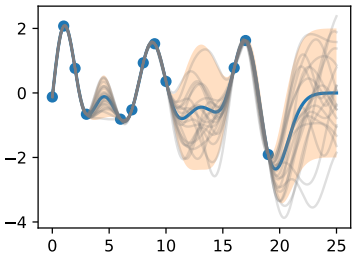


Fig. 21.1.5 Posterior samples, including 95% credible set.

事後サンプルを取り除き、データ、事後平均、95% credible set だけを表示すると、図は少しすっきりする。データから離れるほど不確実性が増えていることに注目しよ。これは epistemic uncertainty の性質である。

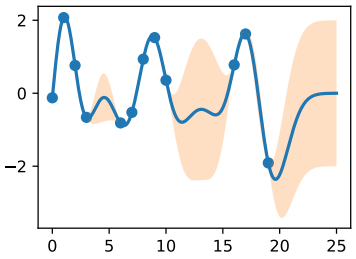


Fig. 21.1.6 Point predictions, and credible set.

データに適合させるために使ったガウス過程の性質は、*covariance function*、別名 *kernel* と呼ばれるものによって強く制御される。ここで使った covariance function は *RBF (Radial Basis Function) kernel* と呼ばれ、次の形をしている。

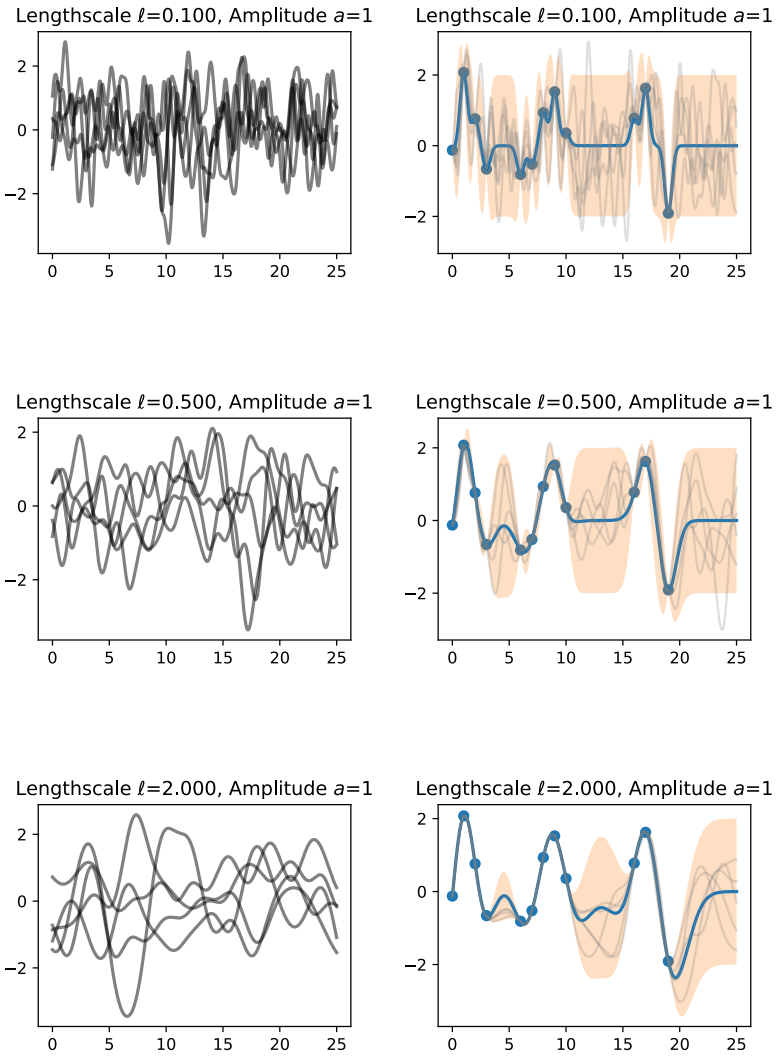
$$k_{\text{RBF}}(x, x') = \text{Cov}(f(x), f(x')) = a^2 \exp\left(-\frac{1}{2\ell^2}||x - x'||^2\right) \tag{21.1.1}$$

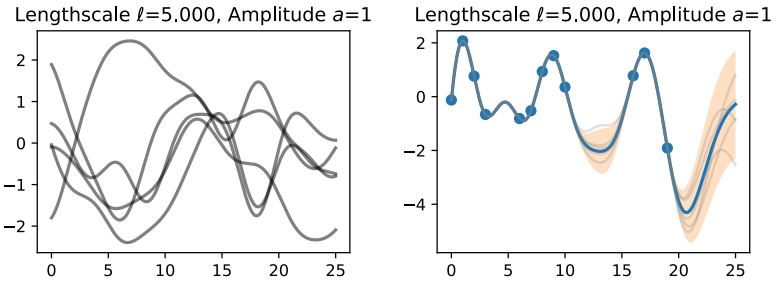
この kernel の *hyperparameters* は解釈可能である。*amplitude* パラメータ a は関数が変化する縦方向のスケールを制御し、*length-scale* パラメータ ℓ は関数の変化率（どれだけ「うねうね」しているか）を制御する。a が大きいほど関数値の振幅は大きくなり、ℓ が大きいほど関数はよりゆっくり変化する。a と ℓ を変えたときに、サンプル事前分布と事後分布の関数がどう変わるか見てみよう。

length-scale は、GP の予測と不確実性に特に顕著な影響を与える。$||x - x'|| = \ell$ のとき、関数値のペア間の共分散は $a^2 \exp(-0.5)$ である。ℓ より大きい距離では、

関数値どうしはほとんど無相関になる。つまり、ある点 x_* で予測したいとき、入力 x が $||x - x'|| > \ell$ を満たす関数値は、予測に強い影響を与えない。

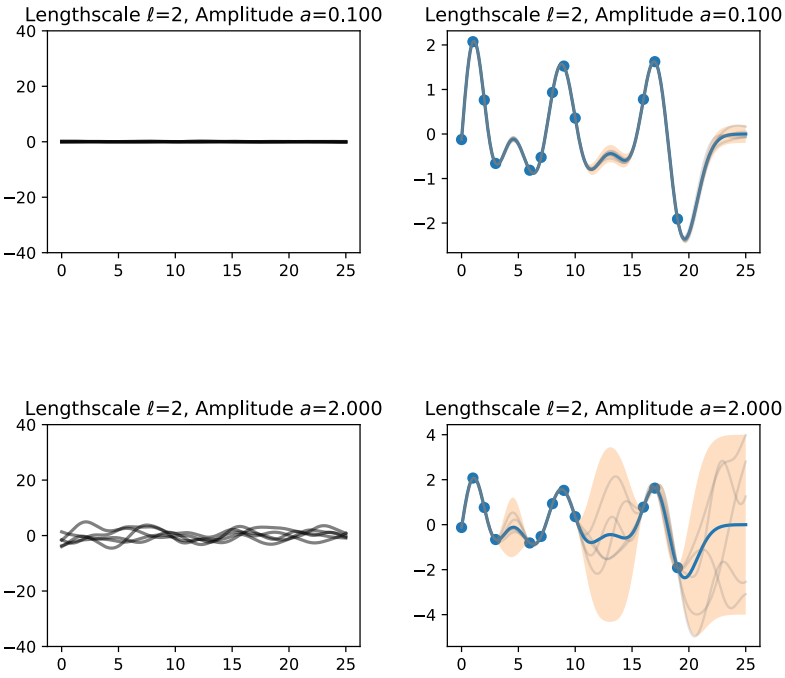
length-scale を変えると、サンプル事前分布・事後分布の関数や credible set がどう変わるか見てみよう。上の適合では length-scale を 2 にしている。ここでは $\ell = 0.1, 0.5, 2, 5, 10$ を考える。length-scale 0.1 は、考えている入力領域の範囲 25 に比べて非常に小さいである。たとえば、$x = 5$ と $x = 10$ における関数値は、このような length-scale では実質的に相関を持たない。一方、length-scale が 10 なら、これらの入力での関数値は強く相関する。以下の図では縦軸スケールが変わっていることに注意しよ。

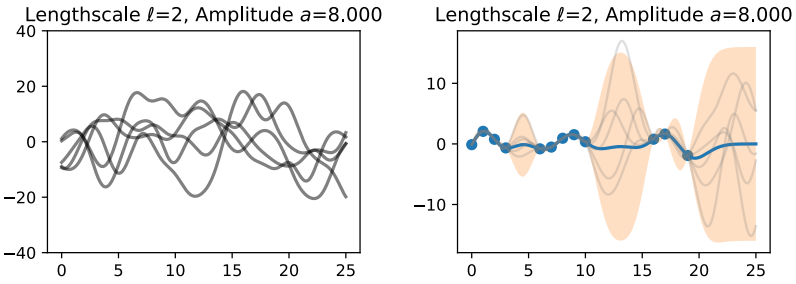




length-scale が大きくなるにつれて、関数の「wiggliness」は減少し、不確実性も減少することがわかる。length-scale が小さいと、データ点が関数値について与える情報が少なくなるため、データから離れるにつれて不確実性は急速に増加する。

次に、length-scale を 2 に固定したまま、amplitude パラメータを変えてみよう。事前サンプルでは縦軸スケールを固定し、事後サンプルでは変化させているので、関数のスケールが大きくなる様子とデータへの適合の両方をはっきり見ることができる。





amplitude パラメータは関数のスケールに影響するが、変化率には影響しないことがわかる。この時点で、私たちの手法の汎化性能は、これらの hyperparameters に妥当な値を与えられるかどうかに依存する、という感覚も得られる。$\ell = 2$ と $a = 1$ の値は妥当な適合を与えているように見えたが、他の値のいくつかはそうではなかった。幸いなことに、*marginal likelihood* と呼ばれるものを使って、これらの hyperparameters を頑健かつ自動的に指定する方法がある。これは推論に関するノートブックで改めて扱う。

では、GP とは実際には何だろうか？先に述べたように、GP は単に、任意の入力集合 $x_1, \ldots, x_n$ によって添字付けられた任意の関数値の集合 $f(x_1), \ldots, f(x_n)$ が、同時多変量ガウス分布に従うと述べるものである。この分布の平均ベクトル μ は *mean function* によって与えられ、通常は定数またはゼロとする。この分布の共分散行列は、すべての入力 x の組に対して kernel を評価したものである。

$$\begin{bmatrix} f(x) \\ f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix} \sim \mathcal{N}\left(\mu, \begin{bmatrix} k(x,x) & k(x,x_1) & \ldots & k(x,x_n) \\ k(x_1,x) & k(x_1,x_1) & \ldots & k(x_1,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_n,x) & k(x_n,x_1) & \ldots & k(x_n,x_n) \end{bmatrix}\right) \tag{21.1.2}$$

式 (21.1.2) は GP の事前分布を指定している。観測した関数値 $f(x_1), \ldots, f(x_n)$ が与えられたとき、任意の x における $f(x)$ の条件付き分布を計算できる。この条件付き分布を *posterior* と呼び、予測に用いる。

特に、

$$f(x)|f(x_1), \ldots, f(x_n) \sim \mathcal{N}(m, s^2) \tag{21.1.3}$$

ここで

$$m = k(x, x_{1:n})k(x_{1:n}, x_{1:n})^{-1} f(x_{1:n}) \tag{21.1.4}$$

$$s^2 = k(x,x) - k(x, x_{1:n})k(x_{1:n}, x_{1:n})^{-1} k(x, x_{1:n}) \tag{21.1.5}$$

である。ここで $k(x, x_{1:n})$ は $i = 1, \ldots, n$ に対して $k(x, x_i)$ を評価して作る $1 \times n$ ベクトルであり、$k(x_{1:n}, x_{1:n})$ は $i, j = 1, \ldots, n$ に対して $k(x_i, x_j)$ を評価して作る $n \times n$ 行列である。m は任意の x に対する点予測として使え、s^2 は不確実性として使う。たとえば、$f(x)$ がその区間に入る確率が 95% となる区間を作りたいなら、$m \pm 2s$ を使う。上のすべての図の予測平均と不確実性は、これらの式を使っ

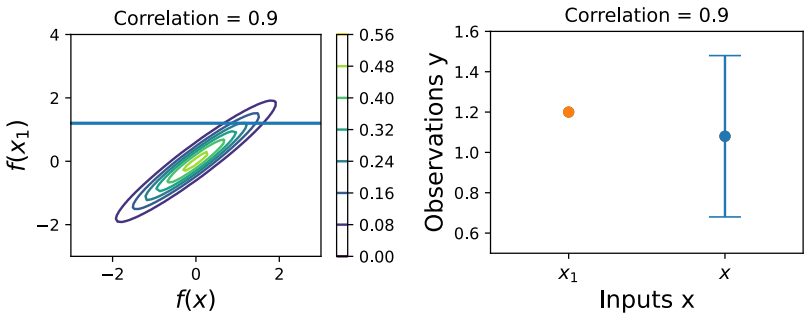
て作成された。観測データ点は $f(x_1), \ldots, f(x_n)$ で与えられ、予測のために細かい刻みの x 点を選んだ。

1 つのデータ点 $f(x_1)$ を観測し、ある x における $f(x)$ の値を求めたいとする。$f(x)$ はガウス過程で記述されるので、$(f(x), f(x_1))$ の同時分布がガウス分布であることがわかる。

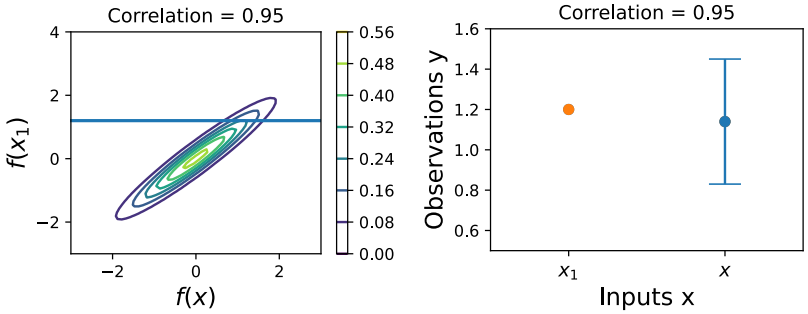
$$\begin{bmatrix} f(x) \\ f(x_1) \end{bmatrix} \sim \mathcal{N}\left(\mu, \begin{bmatrix} k(x,x) & k(x,x_1) \\ k(x_1,x) & k(x_1,x_1) \end{bmatrix}\right) \tag{21.1.6}$$

非対角成分 $k(x,x_1) = k(x_1,x)$ は、関数値がどの程度相関するか—つまり $f(x)$ が $f(x_1)$ からどれだけ強く決まるか—を教えてくれる。すでに見たように、x と x_1 の距離 $||x - x_1||$ に比べて length-scale が大きいと、関数値は強く相関する。$f(x_1)$ から $f(x)$ を決める過程は、関数空間でも、$f(x_1), f(x)$ の同時分布でも可視化できる。まず、$k(x,x_1) = 0.9$、$k(x,x) = 1$ となる x を考える。これは、$f(x)$ の値が $f(x_1)$ の値と中程度に相関していることを意味する。同時分布では、等確率曲線は比較的細い楕円になる。

$f(x_1) = 1.2$ を観測したとする。この $f(x_1)$ の値で条件づけるには、密度の図において 1.2 の位置に水平線を引けばよく、すると $f(x)$ の値は主として $[0.64, 1.52]$ に制約されることがわかる。これを関数空間でも描いており、観測点 $f(x_1)$ をオレンジで、$f(x)$ に対するガウス過程予測分布の 1 標準偏差を青で、平均値 1.08 のまわりに示している。



次に、より強い相関 $k(x,x_1) = 0.95$ があるとする。すると楕円はさらに細くなり、$f(x)$ の値は $f(x_1)$ によってさらに強く決まる。1.2 に水平線を引くと、$f(x)$ の等高線は主として $[0.83, 1.45]$ の値を支持していることがわかる。ここでも、平均予測値 1.14 のまわりに 1 標準偏差を示した関数空間の図を示す。



相関が強くなると、ガウス過程の事後平均予測は 1.2 に近づくことがわかる。また、不確実性（誤差棒）もやや小さくなっている。これらの関数値の間に強い相関があるにもかかわらず、観測したデータ点は 1 つだけなので、不確実性は依然としてかなり大きいままである。

この手順は、観測した点の数にかかわらず、任意の x に対する $f(x)$ の事後分布を与えてくれる。$f(x_1), f(x_2)$ を観測したとしよう。ここでは、特定の $x = x'$ における $f(x)$ の事後分布を関数空間で可視化する。$f(x)$ の厳密な分布は上の式で与えられる。$f(x)$ はガウス分布に従い、平均は

$$m = k(x, x_{1:3})k(x_{1:3}, x_{1:3})^{-1} f(x_{1:3}) \tag{21.1.7}$$

分散は

$$s^2 = k(x, x) - k(x, x_{1:3})k(x_{1:3}, x_{1:3})^{-1} k(x, x_{1:3}) \tag{21.1.8}$$

である。

この入門ノートブックでは、*noise free* の観測を考えてきた。後で見るように、観測ノイズを含めるのは簡単である。データが、潜在的なノイズなし関数 $f(x)$ に iid Gaussian noise $\epsilon(x) \sim \mathcal{N}(0, \sigma^2)$ を加えたものとして生成されると仮定するなら、共分散関数は単に $k(x_i, x_j) \to k(x_i, x_j) + \delta_{ij}\sigma^2$ となる。ここで $\delta_{ij} = 1$ は $i = j$ のとき、そうでなければ 0 である。

ここまでで、ガウス過程を使って解の事前分布と事後分布をどのように指定できるか、そして kernel 関数がこれらの解の性質にどう影響するかについて、ある程度の直感を得た。次のノートブックでは、ガウス過程の事前分布を正確に指定する方法を示し、さまざまな kernel 関数を導入・導出し、その後、kernel の hyperparameters を自動的に学習し、予測のためのガウス過程事後分布を構成する仕組みを説明する。「関数の分布」のような概念に慣れるには時間と練習が必要であるが、GP の予測式を求める実際の手順自体は非常に単純である—そのため、これらの概念について直感的な理解を身につける練習がしやすくなっている。

21.1.1 要約

典型的な機械学習では、何らかの自由パラメータを持つ関数（たとえばニューラルネットワークとその重み）を指定し、そのパラメータの推定に焦点を当て

るが、それらは解釈しにくいことがある。ガウス過程では、その代わりに関数に対する分布そのものについて推論するため、解の高レベルな性質について考えることができる。これらの性質は共分散関数（kernel）によって制御され、そこにはしばしば少数の非常に解釈しやすい hyperparameters がある。これらの hyperparameters には、関数がどれだけ速く（どれだけうねうねと）変化するかを制御する *length-scale* が含まれる。もう 1 つの hyperparameter は amplitude で、関数が変化する縦方向のスケールを制御する。データに適合しうる多様な関数を表現し、それらをすべてまとめて予測分布に統合することは、ベイズ法の特徴的な性質である。データから離れるほど取りうる解のばらつきが大きくなるため、直感的にはデータから離れるにつれて不確実性が増していく。

ガウス過程は、取りうるすべての関数値に対して多変量正規（ガウス）分布を指定することで、関数の分布を表現する。ガウス分布は容易に操作できるので、ある関数値の分布を、他の任意の値の集合に基づいて求めることができる。言い換えると、点の集合を観測したなら、それらの点で条件づけて、他の任意の入力における関数値がどのような分布になりうるかを推論できる。これらの点の相関をどのようにモデル化するかは共分散関数によって決まり、それがガウス過程の汎化特性を定義する。ガウス過程に慣れるには時間がかかるが、扱いやすく、多くの応用があり、ニューラルネットワークのような他のモデルクラスを理解・発展させる助けにもなる。

21.1.2 演習

1. epistemic uncertainty と observation uncertainty の違いは何ですか？

2. 変化率と amplitude 以外に、どのような関数の性質を考慮したいだろうか。また、その性質を持つ関数の現実世界の例にはどのようなものがあるか？

3. ここで考えた RBF 共分散関数は、観測間の共分散（および相関）が入力空間（時刻、空間的位置など）での距離とともに減少すると述べている。これは妥当な仮定だろうか？なぜそう言える、あるいはそう言えないのだろうか？

4. 2 つのガウス変数の和はガウスですか？2 つのガウス変数の積はガウスですか？(a, b) が同時ガウス分布に従うとき、$a|b$（b が与えられたときの a）はガウスですか？ガウスではないですか？

5. $f(x_1) = 1.2$ のデータ点を観測する演習を繰り返せよ。ただし今度は、さらに $f(x_2) = 1.4$ も観測したとする。$k(x, x_1) = 0.9$、$k(x, x_2) = 0.8$ とする。$f(x)$ の値について、$f(x_1)$ だけを観測したときよりも、より確信を持てるだろうか、それともより不確かだろうか？今の $f(x)$ の平均と 95% credible set はどうなるか？

6. 観測ノイズの推定値を増やすと、真の関数の length-scale の推定値は大きくなると思うか、それとも小さくなると思うか？

7. データから離れるにつれて、予測分布の不確実性がある点まで増加した後、増加しなくなるとする。なぜそのようなことが起こるのだろうか？

21.2 ガウス過程の事前分布

ガウス過程（GP）を理解することは、モデル構築と汎化について推論するうえで重要であり、能動学習や深層学習におけるハイパーパラメータ調整を含む、さまざまな応用で最先端の性能を達成するためにも重要である。GP は至るところにあり、それが何であり、どのように使えるのかを知っておくことは私たちにとって有益である。

この節では、関数に対するガウス過程の 事前分布を導入する。次のノートブックでは、これらの事前分布を使って 事後推論を行い、予測を作る方法を示す。次の節は「GP の要点」と見なすことができ、実際にガウス過程を適用するために必要なことを手早く示す。

```
from d2l import torch as d2l
import numpy as np
from scipy.spatial import distance_matrix

d2l.set_figsize()
```

21.2.1 定義

ガウス過程は、任意の有限個を取ると同時ガウス分布に従う確率変数の集合として定義される。関数 $f(x)$ が、平均関数 $m(x)$ と 共分散関数または カーネル $k(x, x')$ をもつガウス過程であるとき、$f(x)$
sim
$mathcalGP(m, k)$ と書く。このとき、任意の入力点 x（時刻、空間位置、画像の画素など）の集合で問い合わせた関数値の集合は、平均ベクトル
mu と共分散行列 K をもつ同時多変量ガウス分布に従う。すなわち、$f(x_1)$,
$dots, f(x_n)$
sim
$mathcalN($
$mu, K)$ であり、ここで
$mu_i = E[f(x_i)] = m(x_i)$、$K_{ij} =$
$textrmCov(f(x_i), f(x_j)) = k(x_i, x_j)$ である。

この定義は抽象的でとっつきにくく見えるかもしれないが、ガウス過程は実際には非常に単純な対象である。任意の関数

$$f(x) = w^\top \phi(x) = \langle w, \phi(x) \rangle, \tag{21.2.1}$$

は、w がガウス（正規）分布から引かれ、
phi が任意の基底関数ベクトル、たとえば
$phi(x) = (1, x, x^2, ..., x^d)^top$ であるとき、ガウス過程である。さらに、任意のガウス過程 $f(x)$ は式 (21.2.1) の形で表せる。まずはガウス過程に少しずつ親しむた

めに、いくつか具体例を見ていこう。そのあとで、それらが実際にはどれほど単純で有用かを理解できるはずである。

21.2.2 単純なガウス過程

$f(x) = w_0 + w_1 x$ とし、$w_0, w_1 \sim \mathcal{N}(0,1)$、さらに w_0, w_1, x はすべて 1 次元であるとする。この関数は、内積 $f(x) = (w_0, w_1)(1, x)^\top$ と等価に書ける。上の (21.2.1) では、$w = (w_0, w_1)^\top$、$\phi(x) = (1, x)^\top$ である。

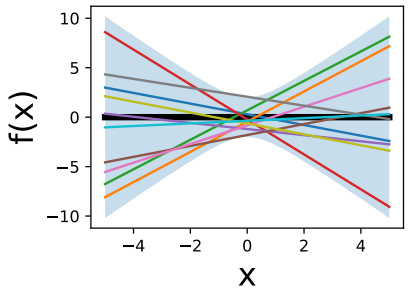
任意の x に対して、$f(x)$ は 2 つのガウス確率変数の和である。ガウス分布は加法に関して閉じているので、$f(x)$ も任意の x に対してガウス確率変数である。実際、任意の特定の x について $f(x)$ は $\mathcal{N}(0, 1+x^2)$ に従うことが計算できる。同様に、任意の入力 $x_1, \ldots, x_n$ に対する関数値の集合 $(f(x_1), \ldots, f(x_n))$ の同時分布は、多変量ガウス分布である。したがって $f(x)$ はガウス過程である。

要するに、$f(x)$ は ランダム関数、あるいは 関数の分布である。w_0, w_1 の値を繰り返しサンプルし、それに対応する関数 $f(x)$ を可視化することで、この分布についていくつかの洞察を得られる。これらは傾きと切片が異なる直線である。以下のようになる。

```
def lin_func(x, n_sample):
    preds = np.zeros((n_sample, x.shape[0]))
    for ii in range(n_sample):
        w = np.random.normal(0, 1, 2)
        y = w[0] + w[1] * x
        preds[ii, :] = y
    return preds

x_points = np.linspace(-5, 5, 50)
outs = lin_func(x_points, 10)
lw_bd = -2 * np.sqrt((1 + x_points ** 2))
up_bd = 2 * np.sqrt((1 + x_points ** 2))

d2l.plt.fill_between(x_points, lw_bd, up_bd, alpha=0.25)
d2l.plt.plot(x_points, np.zeros(len(x_points)), linewidth=4, color='black')
d2l.plt.plot(x_points, outs.T)
d2l.plt.xlabel("x", fontsize=20)
d2l.plt.ylabel("f(x)", fontsize=20)
d2l.plt.show()
```



もし w_0 と w_1 を $\mathcal{N}(0, \alpha^2)$ から引くとしたら、α を変えることが関数分布にどのような影響を与えると考えますか？

21.2.3 重み空間から関数空間へ

上の図では、モデルのパラメータ分布がどのように関数分布を誘導するかを見た。私たちはしばしば、モデル化したい関数が滑らかか、周期的か、急激に変化するかなどについては考えを持っているが、主として解釈しにくいパラメータについて推論するのは比較的面倒である。幸い、ガウス過程は関数について直接推論するための簡単な仕組みを提供する。ガウス分布は第 1・第 2 モーメント、すなわち平均と共分散行列だけで完全に定まるので、ガウス過程も同様に平均関数と共分散関数で定まる。

上の例では、平均関数は

$$m(x) = E[f(x)] = E[w_0 + w_1 x] = E[w_0] + E[w_1]x = 0 + 0 = 0. \tag{21.2.2}$$

同様に、共分散関数は

$$k(x, x') = \mathrm{Cov}(f(x), f(x')) = E[f(x)f(x')] - E[f(x)]E[f(x')] = E[w_0^2 + w_0 w_1 x' + w_1 w_0 x + w_1^2 x x'] = 1 + x x'. \tag{21.2.3}$$

これで、パラメータ分布からサンプルする必要なく、関数分布を直接指定してサンプルできるようになった。たとえば $f(x)$ から引くには、問い合わせたい任意の x の集合に対応する多変量ガウス分布を作り、そこから直接サンプルすればよいのである。この定式化がどれほど有利か、これから見えてくるだろう。

まず、上の単純な直線モデルに対するほぼ同じ導出が、$w \sim \mathcal{N}(u, S)$ をもつ任意の形 $f(x) = w^\top \phi(x)$ のモデルに対して、平均関数と共分散関数を求めるために適用できることに注意する。この場合、平均関数は $m(x) = u^\top \phi(x)$、共分散関数は $k(x, x') = \phi(x)^\top S \phi(x')$ である。$\phi(x)$ は任意の非線形基底関数ベクトルを表せるので、私たちは非常に一般的なモデルクラスを扱っており、無限個のパラメータをもつモデルさえ含まれる。

21.2.4 放射基底関数（RBF）カーネル

放射基底関数（RBF）カーネルは、ガウス過程および一般のカーネル法において最もよく使われる共分散関数である。このカーネルは $k_{\mathrm{RBF}}(x, x') = a^2 \exp\left(-\frac{1}{2\ell^2}||x - x'||^2\right)$ の形をしており、a は振幅パラメータ、ℓ は長さ尺度ハイパーパラメータである。

重み空間から出発してこのカーネルを導出してみよう。次の関数を考える。

$$f(x) = \sum_{i=1}^{J} w_i \phi_i(x), w_i \sim \mathcal{N}\left(0, \frac{\sigma^2}{J}\right), \phi_i(x) = \exp\left(-\frac{(x - c_i)^2}{2\ell^2}\right). \tag{21.2.4}$$

$f(x)$ は、幅 ℓ をもち、点 c_i を中心とする放射基底関数の和であり、次の図に示すようになる。

$f(x)$ は $w = (w_1, \ldots, w_J)^\top$、$\phi(x)$ は各放射基底関数を含むベクトルとして、$w^\top \phi(x)$ の形をしているとみなせる。このガウス過程の共分散関数は

$$k(x, x') = \frac{\sigma^2}{J} \sum_{i=1}^{J} \phi_i(x)\phi_i(x'). \tag{21.2.5}$$

では、パラメータ（および基底関数）の数を無限大にすると何が起こるかを考えよう。$c_J = \log J$、$c_1 = -\log J$、$c_{i+1} - c_i = \Delta c = 2\frac{\log J}{J}$ とし、$J \to \infty$ とする。すると共分散関数はリーマン和になり、

$$k(x, x') = \lim_{J\to\infty} \frac{\sigma^2}{J} \sum_{i=1}^{J} \phi_i(x)\phi_i(x') = \int_{c_0}^{c_\infty} \phi_c(x)\phi_c(x')dc. \tag{21.2.6}$$

$c_0 = -\infty$、$c_\infty = \infty$ とすると、無限個の基底関数を実数直線全体に広げ、それぞれを $\Delta c \to 0$ の間隔で配置することになる。

$$k(x, x') = \int_{-\infty}^{\infty} \exp(-\frac{(x-c)^2}{2\ell^2}) \exp(-\frac{(x'-c)^2}{2\ell^2})dc = \sqrt{\pi}\ell\sigma^2 \exp(-\frac{(x-x')^2}{2(\sqrt{2}\ell)^2}) \propto k_{\text{RBF}}(x, x'). \tag{21.2.7}$$

ここで少し立ち止まって、何をしたのかを噛みしめる価値がある。関数空間表現へ移ることで、無限個のパラメータをもつモデルを、有限の計算量で表現する方法を導出したのである。RBF カーネルをもつガウス過程は 万能近似器であり、任意の連続関数を任意の精度で表現できる。上の導出から、その理由は直感的にも分かる。
ell
*to*0 として各放射基底関数を点質量に潰し、各点質量に好きな高さを与えればよいのである。

したがって、RBF カーネルをもつガウス過程は無限個のパラメータをもつモデルであり、どんな有限のニューラルネットワークよりもはるかに柔軟である。もしかすると、過剰パラメータ化されたニューラルネットワークをめぐる大騒ぎは見当違いなのかもしれない。これから見るように、RBF カーネルをもつ GP は過学習せず、実際、小規模データセットでは特に優れた汎化性能を示す。さらに、(Zhang *et al.*, 2021) にある、ランダムラベル付き画像を完全に当てはめられるのに、構造化された問題では依然としてよく汎化する、といった例は、ガウス過程を用いて完全に再現できる (Wilson and Izmailov, 2020)。ニューラルネットワークは、私たちが思っているほど別物ではないのである。

RBF カーネルをもつガウス過程や、長さ尺度のようなハイパーパラメータについてさらに直感を深めるために、関数分布から直接サンプルしてみよう。これまでと同様、手順は単純である。

1. GP に問い合わせたい入力 x の点 $x_1, \ldots, x_n$ を選ぶ。
2. $m(x_i)$ を $i = 1, \ldots, n$ について、また $k(x_i, x_j)$ を $i, j = 1, \ldots, n$ について評価し、それぞれ平均ベクトルと共分散行列 μ と K を作る。ここで $(f(x_1), \ldots, f(x_n)) \sim \mathcal{N}(\mu, K)$ である。
3. この多変量ガウス分布からサンプルして、サンプル関数値を得る。

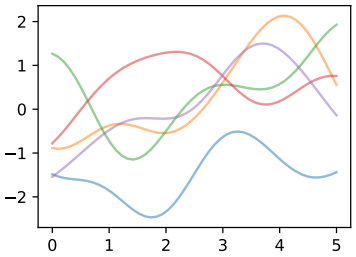
4. さらに何度もサンプルして、それらの点で問い合わせた複数のサンプル関数を可視化する。

この過程を下の図に示す。

```
def rbfkernel(x1, x2, ls=4.):  #@save
    dist = distance_matrix(np.expand_dims(x1, 1), np.expand_dims(x2, 1))
    return np.exp(-(1. / ls / 2) * (dist ** 2))

x_points = np.linspace(0, 5, 50)
meanvec = np.zeros(len(x_points))
covmat = rbfkernel(x_points,x_points, 1)

prior_samples= np.random.multivariate_normal(meanvec, covmat, size=5);
d2l.plt.plot(x_points, prior_samples.T, alpha=0.5)
d2l.plt.show()
```



21.2.5 ニューラルネットワークカーネル

機械学習におけるガウス過程の研究は、ニューラルネットワークの研究によって引き起こされた。ラドフォード・ニールは、ますます大きなベイズニューラルネットワークを追求し、最終的に 1994 年に（後に 1996 年に出版された。これは最も悪名高い NeurIPS の却下例の 1 つだったためである）、無限個の隠れユニットをもつそのようなネットワークが、特定のカーネル関数をもつガウス過程になることを示した (Neal, 1996)。この導出への関心は再燃しており、ニューラル接線カーネルのような考え方が、ニューラルネットワークの汎化特性を調べるために使われている (Matthews *et al.*, 2018) (Novak *et al.*, 2018)。以下のようにしてニューラルネットワークカーネルを導出できる。

1 つの隠れ層をもつニューラルネットワーク関数 $f(x)$ を考える。

$$f(x) = b + \sum_{i=1}^{J} v_i h(x; u_i). \tag{21.2.8}$$

b はバイアス、v_i は隠れ層から出力層への重み、h は任意の有界な隠れユニットの活性化関数、u_i は入力から隠れ層への重み、J は隠れユニット数である。b と v_i は独立で、それぞれ平均 0、分散
$sigma_b^2$ と
$sigma_v^2/J$ をもち、u_i は独立同分布に従うとする。すると中心極限定理を用い

て、任意の関数値の集合 $f(x_1), \ldots, f(x_n)$ が同時多変量ガウス分布に従うことを示せる。

対応するガウス過程の平均関数と共分散関数は次のとおりである。

$$m(x) = E[f(x)] = 0 \tag{21.2.9}$$

$$k(x, x') = \text{cov}[f(x), f(x')] = E[f(x)f(x')] = \\ sigma_b^2 + \frac{1}{J}\sum_{i=1}^{J} \\ sigma_v^2 E[h_i(x; u_i)h_i(x'; u_i)] \tag{21.2.10}$$

場合によっては、この共分散関数をほぼ閉じた形で評価できる。$h(x; u) = \text{erf}(u_0 + \sum_{j=1}^{P} u_j x_j)$、ここで $\text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$、かつ $u \sim \mathcal{N}(0, \Sigma)$ とする。このとき $k(x, x') = \frac{2}{\pi} \sin(\frac{2\tilde{x}^\top \Sigma \tilde{x}'}{\sqrt{(1+2\tilde{x}^\top \Sigma \tilde{x})(1+2\tilde{x}'^\top \Sigma \tilde{x}')}})$ となる。

RBF カーネルは 定常、つまり 平行移動不変であり、したがって $\tau = x - x'$ の関数として書ける。直感的には、定常性とは、入力空間を移動しても、変化率のような関数の高レベルな性質が変わらないことを意味する。一方、ニューラルネットワークカーネルは 非定常である。以下では、このカーネルをもつガウス過程からのサンプル関数を示す。原点付近で関数の見た目が質的に異なることが分かるだろう。

21.2.6 まとめ

ベイズ推論を行う最初のステップは、事前分布を指定することである。ガウス過程は、関数全体にわたる事前分布を指定するために使える。従来の「重み空間」からのモデリングの見方から出発すると、モデルの関数形を先に定め、そのパラメータに分布を導入することで、関数に対する事前分布を誘導できる。あるいは、カーネルによって性質が制御される関数空間で、事前分布を直接指定することもできる。関数空間アプローチには多くの利点がある。実際には無限個のパラメータに対応するモデルを、有限の計算量で構築できるのである！さらに、これらのモデルは非常に柔軟である一方、どのような関数が事前に起こりやすいかについて強い仮定を置くため、小規模データセットでは比較的良い汎化を示す。

関数空間におけるモデルの仮定は、直感的にはカーネルによって制御される。カーネルはしばしば、滑らかさや周期性のような関数の高次の性質を符号化する。多くのカーネルは定常的で、つまり平行移動不変である。定常カーネルをもつガウス過程から引かれた関数は、入力空間のどこを見るかに関係なく、おおむね同じ高レベルの性質（たとえば変化率）をもつ。

ガウス過程は比較的一般的なモデルクラスであり、パラメータにガウス事前分布がある限り、多項式やフーリエ級数など、私たちがすでに知っている多くのモデル例を含む。また、パラメータにガウス分布がなくても、無限個のパラメータをもつニューラルネットワークも含まれる。ラドフォード・ニールによっ

て発見されたこのつながりは、機械学習研究者をニューラルネットワークからガウス過程へと向かわせるきっかけとなった。

21.2.7 演習

1. オーンシュタイン＝ウーレンベック（OU）カーネル $k_{\mathrm{OU}}(x, x') = \exp\left(-\frac{1}{2\ell}||x - x'|\right)$ をもつ GP から、事前関数のサンプルを描いてみなさい。長さ尺度 ℓ を同じに固定したとき、これらの関数は RBF カーネルをもつ GP からのサンプル関数と比べてどのように異なって見えるか？
2. RBF カーネルの 振幅 a^2 を変えると、関数分布にどのような影響があるか？
3. $u(x) = f(x)+2g(x)$ を作るとする。ここで $f(x) \sim \mathcal{GP}(m_1, k_1)$、$g(x) \sim \mathcal{GP}(m_2, k_2)$ である。$u(x)$ はガウス過程か？もしそうなら、その平均関数と共分散関数は何か？
4. $g(x) = a(x)f(x)$ を作るとする。ここで $f(x) \sim \mathcal{GP}(0, k)$、$a(x) = x^2$ である。$g(x)$ はガウス過程か？もしそうなら、その平均関数と共分散関数は何か？$a(x)$ の効果は何か？$g(x)$ から引いたサンプル関数はどのように見えるか？
5. $u(x) = f(x)g(x)$ を作るとする。ここで $f(x) \sim \mathcal{GP}(m_1, k_1)$、$g(x) \sim \mathcal{GP}(m_2, k_2)$ である。$u(x)$ はガウス過程か？もしそうなら、その平均関数と共分散関数は何か？

21.3 ガウス過程推論

この節では、前節で導入した GP 事前分布を用いて、事後推論を行い予測を作成する方法を示す。まずは回帰から始める。回帰では、推論を 閉形式で行うことができる。これは、実際にガウス過程をすぐ使い始めるための「GP の要点」セクションである。まず、基本的な操作をすべてゼロから実装し、その後で GPyTorch[124] を導入する。これにより、最先端のガウス過程を扱ったり、深層ニューラルネットワークと統合したりすることがずっと便利になる。これらのより高度な話題については、次の節で詳しく扱う。その節では、近似推論が必要となる設定　分類、点過程、あるいは非ガウス尤度全般　についても考える。

124



21.3.1 回帰における事後推論

観測モデルは、学習したい関数 $f(x)$ と、観測値 $y(x)$ を結びつける。どちらもある入力 x によって添字付けられる。分類では、x は画像の画素であり、y は対応するクラスラベルである。回帰では、y は通常、地表温度、海面水位、CO_2 濃度などの連続値出力を表す。

回帰では、出力は潜在的なノイズのない関数 $f(x)$ に、独立同分布なガウス雑音

$\epsilon(x)$ が加わったものだと仮定することがよくある。

$$y(x) = f(x) + \epsilon(x), \tag{21.3.1}$$

ここで $\epsilon(x) \sim \mathcal{N}(0, \sigma^2)$ である。$\mathbf{y} = y(X) = (y(x_1), \ldots, y(x_n))^\top$ を訓練観測のベクトル、$\mathbf{f} = (f(x_1), \ldots, f(x_n))^\top$ を、訓練入力 $X = x_1, \ldots, x_n$ で問い合わせた潜在的なノイズのない関数値のベクトルとする。

ここでは $f(x) \sim \mathcal{GP}(m, k)$ を仮定する。これは、任意の関数値の集まり $\mathbf{f}$ が、平均ベクトル $\mu_i = m(x_i)$ と共分散行列 $K_{ij} = k(x_i, x_j)$ をもつ同時多変量ガウス分布に従うことを意味する。RBF カーネル $k(x_i, x_j) = a^2 \exp\left(-\frac{1}{2\ell^2}||x_i - x_j||^2\right)$ は、標準的な共分散関数の選択肢である。記法を簡単にするため、平均関数は $m(x) = 0$ と仮定する。導出は後で容易に一般化できる。

入力の集合

$$X_* = x_{*1}, x_{*2}, \ldots, x_{*m}. \tag{21.3.2}$$

で予測したいとする。すると、x^2 と $p(\mathbf{f}_*|\mathbf{y}, X)$ を求めたいことになる。回帰設定では、$\mathbf{f}_* = f(X_*)$ と $\mathbf{y}$ の同時分布を求めた後、ガウスの恒等式を使ってこの分布を便利に求めることができる。

式 (21.3.1) を訓練入力 X で評価すると、$\mathbf{y} = \mathbf{f} + \boldsymbol{\text{ffl}}$ である。ガウス過程の定義（前節参照）より、$\mathbf{f} \sim \mathcal{N}(0, K(X, X))$ である。ここで $K(X, X)$ は、あり得るすべての入力対 $x_i, x_j \in X$ に対して共分散関数（別名 カーネル）を評価して得られる $n \times n$ 行列である。$\boldsymbol{\text{ffl}}$ は単に $\mathcal{N}(0, \sigma^2)$ からの iid サンプルからなるベクトルなので、分布は $\mathcal{N}(0, \sigma^2 I)$ である。したがって $\mathbf{y}$ は、2 つの独立な多変量ガウス変数の和であり、分布は $\mathcal{N}(0, K(X, X) + \sigma^2 I)$ となる。また、$\text{cov}(\mathbf{f}_*, \mathbf{y}) = \text{cov}(\mathbf{y}, \mathbf{f}_*)^\top = K(X_*, X)$ であることも示せる。ここで $K(X_*, X)$ は、テスト入力と訓練入力のすべての組に対してカーネルを評価して得られる $m \times n$ 行列である。

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(0, \mathbf{A} = \begin{bmatrix} K(X, X) + \sigma^2 I & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix}\right) \tag{21.3.3}$$

その後、標準的なガウスの恒等式を使って、同時分布から条件付き分布を求めることができる（例えば Bishop 第 2 章を参照）。

$\mathbf{f}_*|\mathbf{y}, X, X_* \sim \mathcal{N}(m_*, S_*)$ であり、$m_* = K(X_*, X)[K(X, X) + \sigma^2 I]^{-1}\mathbf{y}$、
$S = K(X_*, X_*) - K(X_*, X)[K(X, X) + \sigma^2 I]^{-1} K(X, X_*)$ である。

通常、予測共分散行列 S 全体を使う必要はなく、各予測の不確実性として S の対角成分だけを使う。そのため、しばしばテスト点の集合ではなく、単一のテスト点 x_* に対する予測分布を書くことが多いである。

カーネル行列には、上で述べた RBF カーネルの振幅 a や長さ尺度 ℓ のように、推定したいパラメータ θ もある。これらの目的のために、周辺尤度 $p(\mathbf{y}|\theta, X)$ を使う。これは、$\mathbf{y}, \mathbf{f}_*$ の同時分布を求めるために周辺分布を導出する過程ですでに得たものである。後で見るように、周辺尤度はモデル適合度とモデル複雑性の項に分解され、ハイパーパラメータ学習におけるオッカムの剃刀の概念を自

動的に組み込みる。詳しくは MacKay Ch. 28 (MacKay, 2003) および Rasmussen and Williams Ch. 5 (Rasmussen and Williams, 2006) を参照しよ。

```
from d2l import torch as d2l
import numpy as np
from scipy.spatial import distance_matrix
from scipy import optimize
import matplotlib.pyplot as plt
import math
import torch
import gpytorch
import os

d2l.set_figsize()
```

21.3.2 GP 回帰における予測とカーネルハイパーパラメータ学習のための式

ここでは、ガウス過程回帰でハイパーパラメータを学習し予測を行う際に使う式をまとめる。繰り返しになるが、入力 $X = \{x_1, \ldots, x_n\}$ によって添字付けられた回帰目標 $\mathbf{y}$ のベクトルがあり、テスト入力 x_* で予測したいとする。分散 σ^2 をもつ独立同分布な加法ゼロ平均ガウス雑音を仮定する。潜在的なノイズのない関数には、平均関数 m とカーネル関数 k をもつガウス過程事前分布 $f(x) \sim \mathcal{GP}(m, k)$ を使う。カーネル自体には、学習したいパラメータ θ がある。例えば RBF カーネル $k(x_i, x_j) = a^2 \exp\left(-\frac{1}{2\ell^2}||x - x'||^2\right)$ を使うなら、$\theta = \{a^2, \ell^2\}$ を学習したいことになる。$K(X, X)$ は、n 個の訓練入力のあり得るすべての組に対してカーネルを評価して得られる $n \times n$ 行列を表す。$K(x_*, X)$ は、$k(x_*, x_i)$ を $i = 1, \ldots, n$ について評価して得られる $1 \times n$ ベクトルを表す。μ は、すべての訓練点 x で平均関数 $m(x)$ を評価して得られる平均ベクトルである。

通常、ガウス過程を扱うときは、2 段階の手順に従う。1. 周辺尤度をこれらのハイパーパラメータに関して最大化することで、カーネルのハイパーパラメータ $\hat{\theta}$ を学習する。2. 予測平均を点予測器として使い、予測標準偏差の 2 倍を用いて 95% の信用集合を作る。ここでは、学習済みハイパーパラメータ $\hat{\theta}$ に条件付ける。

対数周辺尤度は単なる対数ガウス密度であり、次の形をしている:

$$\log p(\mathbf{y}|\theta, X) = -\frac{1}{2}\mathbf{y}^\top [K_\theta(X, X) + \sigma^2 I]^{-1}\mathbf{y} - \frac{1}{2}\log |K_\theta(X, X)| + c \tag{21.3.4}$$

予測分布は次の形をとります:

$$p(y_*|x_*, \mathbf{y}, \theta) = \mathcal{N}(a_*, v_*) \tag{21.3.5}$$

$$a_* = k_\theta(x_*, X)[K_\theta(X, X) + \sigma^2 I]^{-1}(\mathbf{y} - \mu) + \mu \tag{21.3.6}$$

$$v_* = k_\theta(x_*, x_*) - K_\theta(x_*, X)[K_\theta(X, X) + \sigma^2 I]^{-1} k_\theta(X, x_*) \tag{21.3.7}$$

21.3.3 学習と予測の式の解釈

ガウス過程の予測分布について、いくつか重要な点がある。

- モデルクラスは柔軟であるが、GP 回帰では 閉形式で 厳密なベイズ推論を行うことができる。カーネルのハイパーパラメータを学習することを除けば、訓練はない。予測に使う式を正確に書き下せる。この点でガウス過程はかなり例外的であり、その便利さ、多用途性、そして今なお高い人気に大きく貢献している。

- 予測平均 a_* は、訓練目標 $\mathbf{y}$ の線形結合であり、重みはカーネル $k_\theta(x_*, X)[K_\theta(X, X)+\sigma^2 I]^{-1}$ である。後で見るように、カーネル（とそのハイパーパラメータ）はモデルの汎化特性において極めて重要な役割を果たする。

- 予測平均は目標値 $\mathbf{y}$ に明示的に依存するが、予測分散は依存しない。代わりに、予測不確実性は、カーネル関数によって決まるように、テスト入力 x_* が目標位置 X から離れるにつれて増大する。ただし、不確実性は、データから学習されるカーネルハイパーパラメータ θ を通じて、間接的に目標値 $\mathbf{y}$ に依存する。

- 周辺尤度は、モデル適合度とモデル複雑性（行列式の対数）に分解される。周辺尤度は、データと整合的でありつつ最も単純な適合を与えるハイパーパラメータを選びがちである。

- 主な計算ボトルネックは、n 個の訓練点に対する $n \times n$ の対称正定値行列 $K(X, X)$ について、線形方程式を解くことと対数行列式を計算することである。素朴に行うと、これらの操作はそれぞれ $O(n^3)$ の計算量と、カーネル（共分散）行列の各要素に対して $O(n^2)$ のメモリを要し、しばしばコレスキー分解から始める。歴史的には、これらのボトルネックのために GP はおよそ 10,000 点未満の問題に限られてきており、GP は「遅い」という評判を長年持っていたが、これは今ではほぼ 10 年近く不正確である。高度な話題では、GP を何百万点もの問題にスケールさせる方法を議論する。

- よく使われるカーネル関数では、$K(X, X)$ はしばしば特異に近く、コレスキー分解や線形方程式を解くための他の操作で数値的問題を引き起こすことがある。幸い、回帰ではしばしば $K_\theta(X, X) + \sigma^2 I$ を扱うため、ノイズ分散 σ^2 が $K(X, X)$ の対角に加わり、条件数が大幅に改善される。ノイズ分散が小さい場合、あるいはノイズなし回帰を行う場合は、条件数を改善するために対角へ 10^{-6} 程度の小さな“jitter”を加えるのが一般的である。

21.3.4 ゼロからの実例

回帰データを作成し、そのデータを GP で当てはめる。すべての手順をゼロから実装する。次の式からデータをサンプルする。

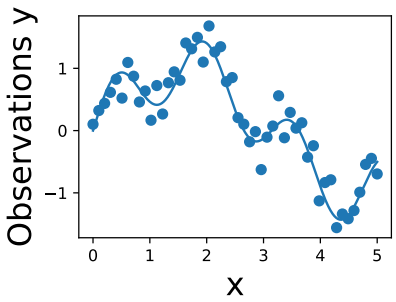
$$y(x) = \sin(x) + \frac{1}{2}\sin(4x) + \epsilon, \tag{21.3.8}$$

ただし $\epsilon \sim \mathcal{N}(0, \sigma^2)$ である。求めたいノイズのない関数は $f(x) = \sin(x) + \frac{1}{2}\sin(4x)$ である。まずはノイズの標準偏差を $\sigma = 0.25$ とする。

```
def data_maker1(x, sig):
    return np.sin(x) + 0.5 * np.sin(4 * x) + np.random.randn(x.shape[0]) * sig

sig = 0.25
train_x, test_x = np.linspace(0, 5, 50), np.linspace(0, 5, 500)
train_y, test_y = data_maker1(train_x, sig=sig), data_maker1(test_x, sig=0.)

d2l.plt.scatter(train_x, train_y)
d2l.plt.plot(test_x, test_y)
d2l.plt.xlabel("x", fontsize=20)
d2l.plt.ylabel("Observations y", fontsize=20)
d2l.plt.show()
```



ここでは、ノイズのある観測が丸印で、求めたい青色のノイズのない関数が見える。

では、潜在的なノイズのない関数 $f(x) \sim \mathcal{GP}(m, k)$ に対して GP 事前分布を指定しよう。平均関数 $m(x) = 0$ と、RBF 共分散関数（カーネル）

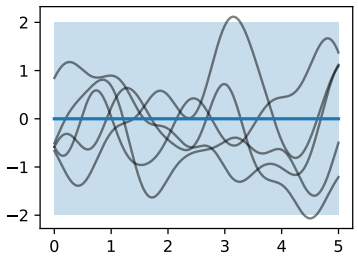
$$k(x_i, x_j) = a^2 \exp\left(-\frac{1}{2\ell^2}||x - x'||^2\right). \tag{21.3.9}$$

を使う。

```
mean = np.zeros(test_x.shape[0])
cov = d2l.rbfkernel(test_x, test_x, ls=0.2)
```

長さ尺度は 0.2 から始めている。データを当てはめる前に、妥当な事前分布を指定できているかを考えることが重要である。この事前分布からのサンプル関数と、95% の信用集合（真の関数がこの領域内にある確率が 95% だと考える）を可視化してみよう。

```
prior_samples = np.random.multivariate_normal(mean=mean, cov=cov, size=5)
d2l.plt.plot(test_x, prior_samples.T, color='black', alpha=0.5)
d2l.plt.plot(test_x, mean, linewidth=2.)
d2l.plt.fill_between(test_x, mean - 2 * np.diag(cov), mean + 2 * np.diag(cov),
                     alpha=0.25)
d2l.plt.show()
```



これらのサンプルは妥当に見えるだろうか。関数の高レベルな性質は、モデル化したいデータの種類と整合しているだろうか。

では、任意のテスト点 x_* における事後予測分布の平均と分散を求める。

$$\bar{f}_* = K(x, x_*)^T (K(x, x) + \sigma^2 I)^{-1} y \tag{21.3.10}$$

$$V(f_*) = K(x_*, x_*) - K(x, x_*)^T (K(x, x) + \sigma^2 I)^{-1} K(x, x_*) \tag{21.3.11}$$

予測を行う前に、カーネルのハイパーパラメータ θ とノイズ分散 σ^2 を学習する必要がある。事前分布の関数が、当てはめるデータに比べて変動が速すぎるように見えたので、長さ尺度の初期値を 0.75 にする。また、ノイズの標準偏差 σ も 0.75 と仮定する。

これらのパラメータを学習するために、これらのパラメータに関して周辺尤度を最大化する。

$$\log p(y|X) = \log \int p(y|f, X) p(f|X) df \tag{21.3.12}$$

$$\log p(y|X) = -\frac{1}{2} y^T (K(x, x) + \sigma^2 I)^{-1} y - \frac{1}{2} \log |K(x, x) + \sigma^2 I| - \frac{n}{2} \log 2\pi \tag{21.3.13}$$

おそらく事前分布の関数は変動が速すぎた。長さ尺度を 0.4 と仮定してみよう。ノイズの標準偏差も 0.75 と仮定する。これらは単なるハイパーパラメータの初期値であり、周辺尤度からこれらのパラメータを学習する。

```
ell_est = 0.4
post_sig_est = 0.5

def neg_MLL(pars):
    K = d2l.rbfkernel(train_x, train_x, ls=pars[0])
    kernel_term = -0.5 * train_y @ \
```

(continues on next page)

(continued from previous page)

```
        np.linalg.inv(K + pars[1] ** 2 * np.eye(train_x.shape[0])) @ train_y
    logdet = -0.5 * np.log(np.linalg.det(K + pars[1] ** 2 * \
                                         np.eye(train_x.shape[0])))
    const = -train_x.shape[0] / 2. * np.log(2 * np.pi)

    return -(kernel_term + logdet + const)


learned_hypers = optimize.minimize(neg_MLL, x0=np.array([ell_est,post_sig_
↪est]),
                                   bounds=((0.01, 10.), (0.01, 10.)))
ell = learned_hypers.x[0]
post_sig_est = learned_hypers.x[1]
```

この例では、長さ尺度 0.299 とノイズの標準偏差 0.24 を学習した。学習されたノイズが真のノイズに非常に近いことに注意しよ。これは、この問題に対して GP が非常によく適合していることを示している。

一般に、カーネルの選択とハイパーパラメータの初期化には慎重な検討が不可欠である。周辺尤度の最適化は初期値に対して比較的頑健であるが、悪い初期化の影響を受けないわけではない。上のスクリプトをさまざまな初期値で実行し、どのような結果になるか試してみよ。

では、これらの学習済みハイパーパラメータで予測を行おう。

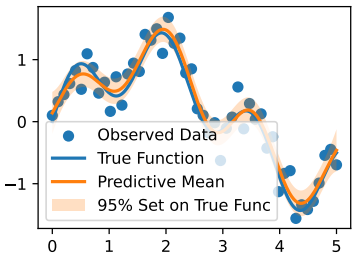
```
K_x_xstar = d2l.rbfkernel(train_x, test_x, ls=ell)
K_x_x = d2l.rbfkernel(train_x, train_x, ls=ell)
K_xstar_xstar = d2l.rbfkernel(test_x, test_x, ls=ell)

post_mean = K_x_xstar.T @ np.linalg.inv((K_x_x + \
                post_sig_est ** 2 * np.eye(train_x.shape[0]))) @ train_y
post_cov = K_xstar_xstar - K_x_xstar.T @ np.linalg.inv((K_x_x + \
                post_sig_est ** 2 * np.eye(train_x.shape[0]))) @ K_x_xstar

lw_bd = post_mean - 2 * np.sqrt(np.diag(post_cov))
up_bd = post_mean + 2 * np.sqrt(np.diag(post_cov))

d2l.plt.scatter(train_x, train_y)
d2l.plt.plot(test_x, test_y, linewidth=2.)
d2l.plt.plot(test_x, post_mean, linewidth=2.)
d2l.plt.fill_between(test_x, lw_bd, up_bd, alpha=0.25)
d2l.plt.legend(['Observed Data', 'True Function', 'Predictive Mean', '95% Set␣
↪on True Func'])
d2l.plt.show()
```

オレンジ色の事後平均は、真のノイズのない関数とほぼ完全に一致していることがわかる。ここで示している 95% の信用集合は、データ点ではなく、潜在的なノイズのない（真の）関数に対するものである。この信用集合は真の関数を完全に含んでおり、広すぎも狭すぎもしないように見える。データ点を含むことは期待していないし、含む必要もない。観測に対する信用集合が欲しいなら、次を計算する。




```
lw_bd_observed = post_mean - 2 * np.sqrt(np.diag(post_cov) + post_sig_est ** 2)
up_bd_observed = post_mean + 2 * np.sqrt(np.diag(post_cov) + post_sig_est ** 2)
```

不確実性には 2 つの源がある。認識論的不確実性は 削減可能な不確実性を表し、偶然的不確実性または 不可約不確実性もある。ここでの 認識論的不確実性は、ノイズのない真の関数値に関する不確実性である。データから離れるにつれてこの不確実性は増大すべきである。なぜなら、データから離れるほど、データと整合する関数値の候補が増えるからである。より多くのデータを観測するにつれて、真の関数に対する信念はより確信を増し、認識論的不確実性は消えていく。この例における 偶然的不確実性は観測ノイズである。データはこのノイズ付きで与えられており、減らすことはできない。

データにおける 認識論的不確実性は、潜在的なノイズのない関数の分散 np.diag(post_cov) によって捉えられる。偶然的不確実性はノイズ分散 post_sig_est**2 によって捉えられる。

残念ながら、不確実性の表し方について人々はしばしば不注意である。多くの論文では、エラーバーがまったく定義されていなかったり、認識論的不確実性と偶然的不確実性のどちらを可視化しているのか、あるいは両方なのかが不明瞭だったり、ノイズ分散とノイズ標準偏差、標準偏差と標準誤差、信頼区間と信用集合などを混同していたりする。不確実性が何を表しているのかを正確にしなければ、それは本質的に無意味である。

不確実性が何を表しているのかに注意を払うという観点から、ここではノイズのない関数の分散推定値の 平方根に *2* 倍を掛けていることが重要である。予測分布はガウス分布なので、この量によって 95% の信用集合を作ることができ、真の関数を 95% の確率で含むと考えられる区間に対する信念を表す。ノイズの分散はまったく別のスケールにあり、はるかに解釈しにくいである。

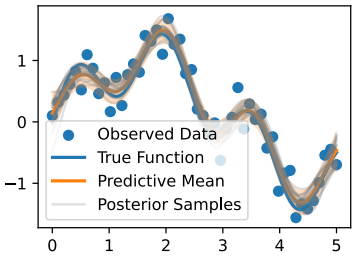
最後に、20 個の事後サンプルを見てみよう。これらのサンプルは、事後的にどのような関数がデータに適合しうると考えているかを示す。

```
post_samples = np.random.multivariate_normal(post_mean, post_cov, size=20)
d2l.plt.scatter(train_x, train_y)
d2l.plt.plot(test_x, test_y, linewidth=2.)
d2l.plt.plot(test_x, post_mean, linewidth=2.)
d2l.plt.plot(test_x, post_samples.T, color='gray', alpha=0.25)
```

(continues on next page)

(continued from previous page)

```
d2l.plt.fill_between(test_x, lw_bd, up_bd, alpha=0.25)
plt.legend(['Observed Data', 'True Function', 'Predictive Mean', 'Posterior␣
↪Samples'])
d2l.plt.show()
```



基本的な回帰アプリケーションでは、事後予測平均と標準偏差を、それぞれ点予測器と不確実性の指標として使うのが最も一般的である。モンテカルロ獲得関数を用いたベイズ最適化や、モデルベース RL のためのガウス過程のような、より高度な応用では、事後サンプルを取る必要があることがよくある。しかし、基本的な応用で厳密には必要でなくても、これらのサンプルはデータへの当てはまりについての直感を与えてくれ、可視化に含めると有用なことが多いである。

21.3.5 GPyTorch で簡単にする

見てきたように、基本的なガウス過程回帰は実際にはゼロからでもかなり簡単に実装できる。しかし、さまざまなカーネルを試したくなったり、近似推論を考えたり（これは分類でも必要である）、GP とニューラルネットワークを組み合わせたり、あるいは 10,000 点程度を超えるデータセットを扱ったりすると、ゼロからの実装は扱いにくく煩雑になる。SKI（KISS-GP とも呼ばれる）のようなスケーラブルな GP 推論の有力な手法の中には、高度な数値線形代数ルーチンを実装する数百行のコードを必要とするものもある。

このような場合、*GPyTorch* ライブラリは大いに役立ちる。GPyTorch については、ガウス過程の数値計算や高度な手法に関する今後のノートブックでさらに詳しく扱う。GPyTorch ライブラリには 多くの例[125] がある。パッケージの雰囲気をつかむために、単純な回帰の例[126] を見て、上の結果を GPyTorch で再現するようにどう適応できるかを示す。これは、上の基本的な回帰を再現するだけにしてはコードが多く見えるかもしれないし、ある意味ではその通りである。しかし、数千行の新しいコードを書く代わりに、下の数行を変えるだけで、さまざまなカーネル、スケーラブルな推論手法、近似推論をすぐに使えるようになる。

125



126



```
# First let's convert our data into tensors for use with PyTorch
train_x = torch.tensor(train_x)
```

(continues on next page)

(continued from previous page)

```
train_y = torch.tensor(train_y)
test_y = torch.tensor(test_y)

# We are using exact GP inference with a zero mean and RBF kernel
class ExactGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(ExactGPModel, self).__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ZeroMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.RBFKernel())

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

このコードブロックでは、データを GPyTorch で使える形式に変換し、厳密推論を使うこと、そして使いたい平均関数（ゼロ）とカーネル関数（RBF）を指定している。例えば gpytorch.kernels.matern_kernel() や gpyotrch.kernels.spectral_mixture_kernel() を呼び出すだけで、他のカーネルも簡単に使える。ここまでで扱ってきたのは厳密推論だけであり、近似を行わずに予測分布を推定できる場合である。ガウス過程では、ガウス尤度がある場合にのみ厳密推論が可能である。より具体的には、観測がガウス過程で表されるノイズのない関数にガウス雑音が加わって生成されると仮定する場合である。今後のノートブックでは、これらの仮定が成り立たない分類などの他の設定を扱う。

```
# Initialize Gaussian likelihood
likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = ExactGPModel(train_x, train_y, likelihood)
training_iter = 50
# Find optimal model hyperparameters
model.train()
likelihood.train()
# Use the adam optimizer, includes GaussianLikelihood parameters
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)
# Set our loss as the negative log GP marginal likelihood
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)
```

ここでは、使いたい尤度（ガウス）、カーネルのハイパーパラメータを学習するために使う目的関数（ここでは周辺尤度）、そしてその目的関数を最適化するために使う手順（この場合は Adam）を明示的に指定している。Adam は「確率的」最適化手法であるが、この場合はフルバッチ Adam であることに注意しよ。周辺尤度はデータごとに因数分解できないため、データの「ミニバッチ」に対する最適化器を使っても収束が保証されない。L-BFGS のような他の最適化器も GPyTorch でサポートされている。標準的な深層学習とは異なり、周辺尤度をうまく最適化できることは良い汎化と強く結びついているため、計算コストが高すぎない限り、L-BFGS のような強力な最適化器を使いたくなることが多いである。

```
for i in range(training_iter):
    # Zero gradients from previous iteration
    optimizer.zero_grad()
    # Output from model
    output = model(train_x)
    # Calc loss and backprop gradients
    loss = -mll(output, train_y)
    loss.backward()
    if i % 10 == 0:
        print(f'Iter {i+1:d}/{training_iter:d} - Loss: {loss.item():.3f} '
              f'squared lengthscale: '
              f'{model.covar_module.base_kernel.lengthscale.item():.3f} '
              f'noise variance: {model.likelihood.noise.item():.3f}')
    optimizer.step()
```

```
Iter 1/50 - Loss: 0.982 squared lengthscale: 0.693 noise variance: 0.693
Iter 11/50 - Loss: 0.696 squared lengthscale: 0.519 noise variance: 0.313
Iter 21/50 - Loss: 0.441 squared lengthscale: 0.535 noise variance: 0.128
Iter 31/50 - Loss: 0.336 squared lengthscale: 0.521 noise variance: 0.057
Iter 41/50 - Loss: 0.353 squared lengthscale: 0.517 noise variance: 0.042
```

ここで実際に最適化手順を実行し、10 イテレーションごとに損失の値を出力している。

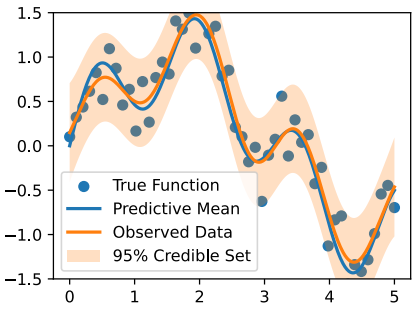
```
# Get into evaluation (predictive posterior) mode
test_x = torch.tensor(test_x)
model.eval()
likelihood.eval()
observed_pred = likelihood(model(test_x))
```

上のコードブロックにより、テスト入力に対して予測を行えるようになる。

```
with torch.no_grad():
    # Initialize plot
    f, ax = d2l.plt.subplots(1, 1, figsize=(4, 3))
    # Get upper and lower bounds for 95\% credible set (in this case, in
    # observation space)
    lower, upper = observed_pred.confidence_region()
    ax.scatter(train_x.numpy(), train_y.numpy())
    ax.plot(test_x.numpy(), test_y.numpy(), linewidth=2.)
    ax.plot(test_x.numpy(), observed_pred.mean.numpy(), linewidth=2.)
    ax.fill_between(test_x.numpy(), lower.numpy(), upper.numpy(), alpha=0.25)
    ax.set_ylim([-1.5, 1.5])
    ax.legend(['True Function', 'Predictive Mean', 'Observed Data',
               '95% Credible Set'])
```

最後に、当てはまりを描画する。

当てはまりは事実上同一であることがわかる。いくつか注意点がある。GPyTorch は 二乗された長さ尺度と観測ノイズを扱っている。例えば、ゼロから書いたコードで学習されたノイズ標準偏差は約 0.283 でした。GPyTorch が見つけた



ノイズ分散は $0.81 \approx 0.283^2$ である。GPyTorch の図では、潜在関数空間ではなく観測空間における信用集合も示しており、実際に観測データ点を覆っていることを示している。

21.3.6 まとめ

ガウス過程事前分布とデータを組み合わせて事後分布を作り、それを予測に使うことができる。また、周辺尤度を作ることもでき、これはガウス過程の変動速度などの性質を制御するカーネルハイパーパラメータの自動学習に役立つ。回帰における事後の構成とカーネルハイパーパラメータの学習の仕組みは単純で、コードはおよそ十数行で済む。このノートブックは、ガウス過程をすぐに「使い始めたい」読者にとって良い参考資料である。また、GPyTorch ライブラリも紹介した。基本的な回帰のための GPyTorch コードは比較的長いものの、他のカーネル関数や、今後のノートブックで扱うより高度な機能　スケーラブル推論や分類のための非ガウス尤度など　へは、数行変えるだけで簡単に拡張できる。

21.3.7 演習

1. カーネルハイパーパラメータを 学習することの重要性と、ハイパーパラメータやカーネルがガウス過程の汎化特性に与える影響を強調してきた。ハイパーパラメータを学習する手順を飛ばし、代わりにさまざまな長さ尺度とノイズ分散を仮定して、予測への影響を確認してみせよ。長さ尺度が大きいとどうなるか。小さいとどうなるか。ノイズ分散が大きいとどうなるか。小さいとどうなるか。

2. 周辺尤度は凸最適化問題ではないが、長さ尺度やノイズ分散のようなハイパーパラメータは GP 回帰で信頼性高く推定できる、と述べた。これは一般に正しいである。実際、周辺尤度は、経験的自己相関関数（「コビアログラム」）を当てはめる空間統計学の従来手法よりも、長さ尺度ハイパーパラメータの学習に はるかに優れている。少なくとも最近のスケーラブル推論の研究以前において、機械学習がガウス過程研究に与えた最大の貢献は、ハイパーパラメータ学習のための周辺尤度の導入だったと言えるだろう。

しかし、これらのパラメータの組み合わせが異なるだけで、多くのデータセッ

トに対して解釈可能でもっともらしい説明が異なり、目的関数に局所最適が生じる。長さ尺度が大きい場合、真の基礎関数はゆっくり変化すると仮定していることになる。観測データが 実際に大きく変動しているなら、大きな長さ尺度を正当化できるのは、大きなノイズ分散がある場合だけである。逆に、長さ尺度が小さい場合、当てはまりはデータの変動に非常に敏感になり、ノイズ（偶然的不確実性）で変動を説明する余地がほとんどなくなる。

これらの局所最適を見つけられるか試してみよ。大きな長さ尺度と大きなノイズ、そして小さな長さ尺度と小さなノイズで初期化してみよう。異なる解に収束するか？

3. ベイズ法の根本的な利点の 1 つは、認識論的不確実性を自然に表現できることだと述べた。上の例では、認識論的不確実性の効果を完全には見ることができない。代わりに `test_x = np.linspace(0, 10, 1000)` で予測してみよ。予測がデータを超えて進むにつれて、95% の信用集合はどうなるか。その区間で真の関数を覆いますか。その領域で偶然的不確実性だけを可視化するとどうなるか。

4. 上の例を、訓練点数を 10,000、20,000、40,000 にして実行し、実行時間を測定してみよ。訓練時間はどのようにスケールするか。あるいは、テスト点数に対して実行時間はどうスケールするか。予測平均と予測分散で違いはあるか。理論的に訓練・テスト時間の計算量を求めることと、上のコードを異なる点数で実行することの両方で答えよ。

5. GPyTorch の例を、Matern カーネルなど異なる共分散関数で実行してみよ。結果はどう変わりますか。GPyTorch ライブラリにある spectral mixture カーネルはどうだろうか。周辺尤度で学習しやすいものとそうでないものはあるか。長距離予測と短距離予測で有用性に違いはあるか。

6. GPyTorch の例では観測ノイズを含めた予測分布を描いたが、ゼロからの例では認識論的不確実性だけを含めた。GPyTorch の例をやり直し、今度は認識論的不確実性だけを描画して、ゼロからの結果と比較せよ。予測分布は同じように見えるか？（同じはずである。）

22 ハイパーパラメータ最適化

Aaron Klein (*Amazon*), **Matthias Seeger** (*Amazon*), and **Cedric Archambeau** (*Amazon*)

あらゆる機械学習モデルの性能は、そのハイパーパラメータに依存する。ハイパーパラメータは、学習アルゴリズムや基礎となる統計モデルの構造を制御する。しかし、実際にハイパーパラメータを選ぶための一般的な方法はない。そのため、ハイパーパラメータは試行錯誤で設定されることが多く、場合によっては実務者がデフォルト値のままにしてしまうこともあり、その結果、汎化性能が最適でないものになりがちである。

ハイパーパラメータ最適化は、この問題に対して体系的なアプローチを提供する。これは、この問題を最適化問題として定式化することで実現される。良いハイパーパラメータの組は、少なくとも検証誤差を最小化すべきである。機械学習で生じる他の多くの最適化問題と比べると、ハイパーパラメータ最適化は入れ子構造の最適化であり、各反復で機械学習モデルの学習と検証が必要になる。

この章では、まずハイパーパラメータ最適化の基礎を紹介する。また、元の目的関数の安価に評価できる代理を活用することで、ハイパーパラメータ最適化全体の効率を改善する最近の進展についても説明する。この章の終わりには、自分自身の機械学習アルゴリズムのハイパーパラメータを最適化するために、最先端のハイパーパラメータ最適化手法を適用できるようになっているはずである。

22.1 ハイパーパラメータ最適化とは何か？

前の章で見てきたように、深層ニューラルネットワークには、学習中に獲得される多数のパラメータ、すなわち重みがある。これに加えて、すべてのニューラルネットワークには、ユーザーが設定しなければならない追加の ハイパーパラメータがある。たとえば、確率的勾配降下法が訓練損失の局所最適に収束するようにするには（Section 15 を参照）、学習率とバッチサイズを調整する必要がある。訓練データセットへの過学習を避けるために、重み減衰（Section 6.7 を参照）やドロップアウト（Section 8.6 を参照）のような正則化パラメータを設定しなければならないこともある。層数や各層あたりのユニット数またはフィル

ター数（すなわち、実効的な重みの数）を設定することで、モデルの容量と帰納バイアスを定義できる。

残念ながら、これらのハイパーパラメータを訓練損失の最小化によって単純に調整することはできない。そうすると訓練データへの過学習を招くからである。たとえば、ドロップアウトや重み減衰のような正則化パラメータをゼロに設定すると訓練損失は小さくなるが、汎化性能を損なう可能性がある。

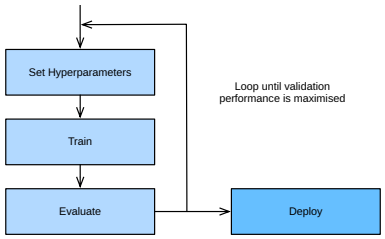


Fig. 22.1.1 異なるハイパーパラメータでモデルを複数回訓練することから成る、機械学習における典型的なワークフロー。

別の形の自動化がなければ、ハイパーパラメータは試行錯誤によって手動で設定しなければならず、機械学習ワークフローの中でも時間がかかり難しい部分になる。たとえば、CIFAR-10 上で ResNet（Section 11.6 を参照）を学習する場合を考えてみよう。Amazon Elastic Cloud Compute (EC2) の `g4dn.xlarge` インスタンスでは 2 時間以上かかる。たとえ 10 通りのハイパーパラメータ設定を順番に試すだけでも、これだけでおよそ 1 日かかってしまう。さらに悪いことに、ハイパーパラメータは通常、アーキテクチャやデータセットをまたいでそのまま転用できず (Bardenet *et al.*, 2013, Feurer *et al.*, 2022, Wistuba *et al.*, 2018)、新しいタスクごとに再最適化する必要がある。また、ほとんどのハイパーパラメータには経験則がなく、妥当な値を見つけるには専門知識が必要である。

ハイパーパラメータ最適化（*HPO*）アルゴリズムは、この問題に原理的かつ自動化された方法で取り組むよう設計されている (Feurer and Hutter, 2018)。これは、これを大域最適化問題として定式化することで実現される。標準的な目的関数はホールドアウトした検証データセット上の誤差であるが、原理的には他のビジネス指標でも構わない。これに、学習時間、推論時間、モデルの複雑さなどの副次的な目的を組み合わせたり、制約として課したりできる。

近年、ハイパーパラメータ最適化は ニューラルアーキテクチャ探索（*NAS*）(Elsken *et al.*, 2018, Wistuba *et al.*, 2019) に拡張され、まったく新しいニューラルネットワークアーキテクチャを見つけることが目標になっている。従来の HPO と比べると、NAS は計算コストがさらに高く、実用上成立させるためには追加の工夫が必要である。HPO と NAS の両方は、ML パイプライン全体の自動化を目指す AutoML (Hutter *et al.*, 2019) の下位分野とみなせる。

この節では HPO を紹介し、Section 7.5 で導入したロジスティック回帰の例に対して、最適なハイパーパラメータを自動的に見つける方法を示す。

22.1.1 最適化問題

まずは簡単な玩具問題から始めよう。Fashion MNIST データセット上で、Section 7.5 の多クラスロジスティック回帰モデル `SoftmaxRegression` の学習率を探索し、検証誤差を最小化する。バッチサイズやエポック数のような他のハイパーパラメータも調整する価値はあるが、簡単のためここでは学習率のみに注目する。

```
from d2l import torch as d2l
import numpy as np
import torch
from torch import nn
from scipy import stats
```

```
config_space = {"learning_rate": stats.loguniform(1e-4, 1)}
```


```
config_space = {"learning_rate": stats.loguniform(1e-4, 1)}
```

```
config_space = {"learning_rate": stats.loguniform(1e-4, 1)}
```


HPO を実行する前に、まず 2 つの要素を定義する必要がある。目的関数と構成空間である。

目的関数

学習アルゴリズムの性能は、ハイパーパラメータ空間 $\mathbf{x} \in \mathcal{X}$ から検証損失へ写像する関数 $f : \mathcal{X} \to \mathbb{R}$ とみなせる。$f(\mathbf{x})$ を評価するたびに、機械学習モデルを訓練して検証しなければならず、深層ニューラルネットワークを大規模データセットで学習する場合には時間と計算資源を大量に消費する。基準 $f(\mathbf{x})$ が与えられたとき、目標は $\mathbf{x}_\star \in \operatorname{argmin}_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x})$ を見つけることである。

f を $\mathbf{x}$ に関して微分する簡単な方法はない。なぜなら、勾配を学習プロセス全体を通して伝播させる必要があるからである。近似的な「ハイパー勾配」によって HPO を駆動する最近の研究 (Franceschi *et al.*, 2017, Maclaurin *et al.*, 2015) はあるが、既存手法はいずれもまだ最先端手法に競争力があるとは言えず、ここでは扱わない。さらに、f の評価に伴う計算負荷のため、HPO アルゴリズムはできるだけ少ないサンプルで大域最適に近づく必要がある。

ニューラルネットワークの学習は確率的である（たとえば、重みはランダムに初期化され、ミニバッチはランダムにサンプリングされる）ため、観測値にはノイズが含まれる：$y \sim f(\mathbf{x}) + \epsilon$。通常、観測ノイズ $\epsilon \sim N(0, \sigma)$ はガウス分布に従うと仮定する。

これらすべての課題に直面した場合、通常は大域最適を厳密に追求するのではなく、性能の良いハイパーパラメータ設定を少数、迅速に見つけることを目指す。しかし、多くのニューラルネットワークモデルは計算要求が大きいため、数日から数週間の計算時間が必要となる場合がある。Section 22.4 では、探索を

分散させるか、目的関数のより安価に評価できる近似を用いることで、最適化プロセスをどのように高速化できるかを見ていく。

まずは、モデルの検証誤差を計算する方法から始める。

```
class HPOTrainer(d2l.Trainer):  #@save
    def validation_error(self):
        self.model.eval()
        accuracy = 0
        val_batch_idx = 0
        for batch in self.val_dataloader:
            with torch.no_grad():
                x, y = self.prepare_batch(batch)
                y_hat = self.model(x)
                accuracy += self.model.accuracy(y_hat, y)
            val_batch_idx += 1
        return 1 -  accuracy / val_batch_idx
```

```
errors, values = [], []
num_iterations = 5

for i in range(num_iterations):
    learning_rate = config_space["learning_rate"].rvs()
    print(f"Trial {i}: learning_rate = {learning_rate}")
    y = hpo_objective_softmax_classification({"learning_rate": learning_rate})
    print(f"    validation_error = {y}")
    values.append(learning_rate)
    errors.append(y)
```


```
errors, values = [], []
num_iterations = 5

for i in range(num_iterations):
    learning_rate = config_space["learning_rate"].rvs()
    print(f"Trial {i}: learning_rate = {learning_rate}")
    y = hpo_objective_softmax_classification({"learning_rate": learning_rate})
    print(f"    validation_error = {y}")
    values.append(learning_rate)
    errors.append(y)
```

```
errors, values = [], []
num_iterations = 5

for i in range(num_iterations):
    learning_rate = config_space["learning_rate"].rvs()
    print(f"Trial {i}: learning_rate = {learning_rate}")
    y = hpo_objective_softmax_classification({"learning_rate": learning_rate})
    print(f"    validation_error = {y}")
    values.append(learning_rate)
    errors.append(y)
```


ここでは、ハイパーパラメータ構成 `config` に含まれる `learning_rate` に関し

て検証誤差を最適化する。各評価では、max_epochs エポックだけモデルを訓練し、その後で検証誤差を計算して返す。

```
def hpo_objective_softmax_classification(config, max_epochs=8):
    learning_rate = config["learning_rate"]
    trainer = d2l.HPOTrainer(max_epochs=max_epochs)
    data = d2l.FashionMNIST(batch_size=16)
    model = d2l.SoftmaxRegression(num_outputs=10, lr=learning_rate)
    trainer.fit(model=model, data=data)
    return d2l.numpy(trainer.validation_error())
```

```
best_idx = np.argmin(errors)
print(f"optimal learning rate = {values[best_idx]}")
```


```
best_idx = np.argmin(errors)
print(f"optimal learning rate = {values[best_idx]}")
```

```
best_idx = np.argmin(errors)
print(f"optimal learning rate = {values[best_idx]}")
```


構成空間

目的関数 $f(\mathbf{x})$ に加えて、最適化対象となる実行可能集合 $\mathbf{x} \in \mathcal{X}$、すなわち 構成空間または 探索空間も定義する必要がある。ロジスティック回帰の例では、次を用いる。

```
config_space = {"learning_rate": stats.loguniform(1e-4, 1)}
```

ここでは SciPy の loguniform オブジェクトを使っている。これは対数空間で -4 から -1 の間の一様分布を表す。このオブジェクトを使うと、この分布から乱数をサンプリングできる。

各ハイパーパラメータには、learning_rate のような float といったデータ型に加えて、閉じた有界範囲（すなわち下限と上限）がある。通常は、各ハイパーパラメータに対して事前分布（たとえば一様分布や対数一様分布）を割り当て、そこからサンプリングする。learning_rate のような正のパラメータの中には、最適値が数桁にわたって変わりうるため対数スケールで表すのが最適なものがある。一方、モメンタムのようなものは線形スケールで扱いる。

以下では、多層パーセプトロンの典型的なハイパーパラメータからなる構成空間の簡単な例を、その型と標準的な範囲とともに示す。

多層パーセプトロンの構成空間の例

Table 22.1.1: label:tab_example_configspace

Name	Type	Hyperparameter Ranges	log-scale
learning rate	float	$[10^{-6}, 10^{-1}]$	yes
batch size	integer	$[8, 256]$	yes
momentum	float	$[0, 0.99]$	no
activation function	categorical	{tanh, relu}	•
number of units	integer	$[32, 1024]$	yes
number of layers	integer	$[1, 6]$	no

一般に、構成空間
$mathcal{X}$ の構造は複雑で、
$mathbb{R}^d$ とはかなり異なることがある。実際には、あるハイパーパラメータが他の値に依存する場合がある。たとえば、多層パーセプトロンの層数と、各層のユニット数を調整しようとしているとする。l
$textrm{−}th$ 層のユニット数は、ネットワークが少なくとも $l+1$ 層を持つ場合にのみ意味を持つ。このような高度な HPO 問題は本章の範囲を超えている。興味のある読者は (Baptista and Poloczek, 2018, Hutter *et al.*, 2011, Jenatton *et al.*, 2017) を参照されたい。

構成空間はハイパーパラメータ最適化において重要な役割を果たする。なぜなら、構成空間に含まれていないものを見つけられるアルゴリズムは存在しないからである。一方で、範囲が広すぎると、性能の良い構成を見つけるための計算予算が現実的でなくなることがある。

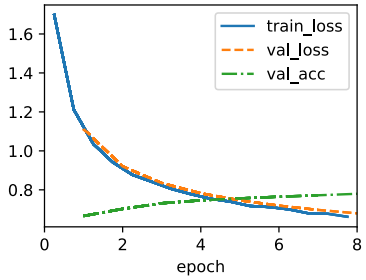
22.1.2 ランダム探索

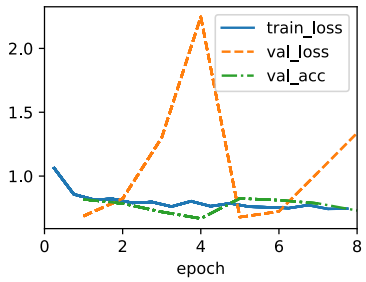
ランダム探索は、最初に扱うハイパーパラメータ最適化アルゴリズムである。ランダム探索の主な考え方は、あらかじめ定めた予算（たとえば最大反復回数）を使い切るまで構成空間から独立にサンプリングし、観測された中で最良の構成を返すことである。すべての評価は独立に並列実行できるが（Section 22.3 を参照）、ここでは簡単のため逐次ループを使う。

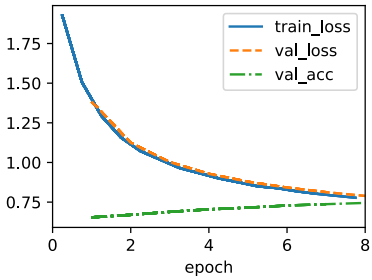
```
errors, values = [], []
num_iterations = 5

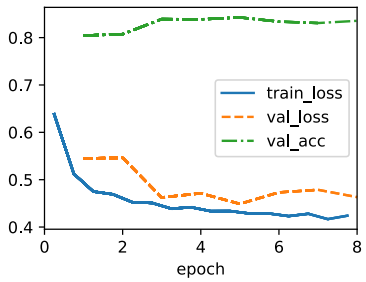
for i in range(num_iterations):
    learning_rate = config_space["learning_rate"].rvs()
    print(f"Trial {i}: learning_rate = {learning_rate}")
    y = hpo_objective_softmax_classification({"learning_rate": learning_rate})
    print(f"    validation_error = {y}")
    values.append(learning_rate)
    errors.append(y)
```

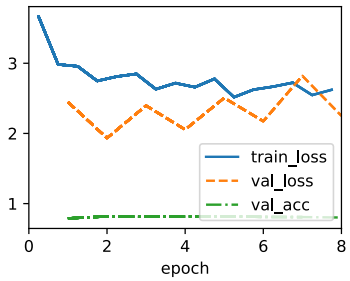
```
    validation_error = 0.19819998741149902
```


train_loss
val_loss
val_acc
1.6
1.4
1.2
1.0
0.8
0
2
4
6
8
epoch


train_loss
val_loss
val_acc
2.0
1.5
1.0
0
2
4
6
8
epoch


train_loss
val_loss
val_acc
1.75
1.50
1.25
1.00
0.75
0
2
4
6
8
epoch


train_loss
val_loss
val_acc
0.8
0.7
0.6
0.5
0.4
0
2
4
6
8
epoch



最良の学習率は、単に検証誤差が最も低いものである。

```
best_idx = np.argmin(errors)
print(f"optimal learning rate = {values[best_idx]}")
```

```
optimal learning rate = 0.046134071504431153
```

その単純さと汎用性のため、ランダム探索は最も頻繁に使われる HPO アルゴリズムの 1 つである。洗練された実装を必要とせず、各ハイパーパラメータに対する確率分布を定義できる限り、どのような構成空間にも適用できる。

残念ながら、ランダム探索にはいくつかの欠点もある。第 1 に、これまでに得られた観測結果に基づいてサンプリング分布を適応させない。そのため、性能の悪い構成をサンプルする確率も、性能の良い構成をサンプルする確率も同じである。第 2 に、初期性能が悪く、これまでに見つかった構成を上回る可能性が低いものにも、すべて同じ量の資源を費やしてしまう。

次の節では、モデルを使って探索を導くことでランダム探索の欠点を克服する、よりサンプル効率の高いハイパーパラメータ最適化アルゴリズムを見ていく。また、性能の悪い構成の評価を自動的に打ち切って最適化を高速化するアルゴリズムも見ていく。

22.1.3 まとめ

この節では、ハイパーパラメータ最適化（HPO）を導入し、構成空間と目的関数を定義することで、それを大域最適化として定式化できることを示した。また、最初の HPO アルゴリズムであるランダム探索を実装し、単純なソフトマックス分類問題に適用した。

ランダム探索は非常に単純であるが、固定されたハイパーパラメータ集合をそのまま評価するグリッドサーチよりも優れた選択肢である。ランダム探索は、次元の呪い (Bellman, 1966) をある程度緩和し、基準がハイパーパラメータの小さな部分集合に最も強く依存している場合には、グリッドサーチよりはるかに効率的になりえる。

22.1.4 演習

1. この章では、互いに重ならない訓練集合で学習した後のモデルの検証誤差を最適化する。簡単のため、コードでは `Trainer.val_dataloader` を使っており、これは `FashionMNIST.val` のローダーに対応している。

   1. （コードを見て）これが、元の FashionMNIST の訓練集合（60000 例）を訓練に、元の テスト集合（10000 例）を検証に使っていることを確認せよ。

   2. なぜこのやり方は問題になりうるのだろうか？ヒント：Section 6.6、特にモデル選択について読み直せよ。

   3. 代わりに何をすべきだっただろうか？

2. 上で、勾配降下法によるハイパーパラメータ最適化は非常に難しいと述べた。たとえば、FashionMNIST データセット（Section 8.2）上で 2 層パーセプトロンをバッチサイズ 256 で学習する小さな問題を考えてみよう。1 エポック学習した後の検証指標を最小化するために、SGD の学習率を調整したいとする。

   1. なぜこの目的のために検証 誤差を使えないのだろうか？検証セット上ではどの指標を使うべきだろうか？

   2. 1 エポック学習した後の検証指標の計算グラフを、おおまかに描け。初期重みとハイパーパラメータ（たとえば学習率）がこのグラフへの入力ノードであると仮定して構わない。ヒント：Section 8.3 の計算グラフについて読み直せよ。

   3. このグラフ上での順伝播中に保存する必要がある浮動小数点値の数を、おおまかに見積もれ。ヒント：FashionMNIST には 60000 例ある。必要なメモリは各層の後の活性化で支配されると仮定し、層の幅は Section 8.2 で確認せよ。

   4. 必要な計算量と保存容量が膨大であること以外に、勾配ベースのハイパーパラメータ最適化にはどのような問題があるだろうか？ヒント：Section 8.4 の勾配消失と勾配爆発について読み直せ。

   5. 発展: 勾配ベース HPO に対する洗練された（とはいえまだやや実用的ではない）アプローチについて (Maclaurin *et al.*, 2015) を参照せよ。

3. グリッドサーチも別の HPO ベースラインである。各ハイパーパラメータについて等間隔のグリッドを定義し、その（組合せ的な）直積を順にたどって構成を提案する。

   1. 上で、基準がハイパーパラメータの小さな部分集合に最も強く依存している場合、十分に多くのハイパーパラメータに対する HPO ではランダム探索がグリッドサーチよりはるかに効率的になりうると述べた。なぜだろうか？ヒント：(Bergstra *et al.*, 2011) を参照せよ。

22.2 ハイパーパラメータ最適化 API

方法論に入る前に、まずはさまざまな HPO アルゴリズムを効率よく実装できる基本的なコード構造について説明する。一般に、ここで扱うすべての HPO アルゴリズムは、探索と スケジューリングという 2 つの意思決定プリミティブを実装する必要がある。まず、新しいハイパーパラメータ構成をサンプリングする必要がある。これは多くの場合、構成空間に対する何らかの探索を伴いる。次に、各構成について、その評価をいつ行うかをスケジュールし、どれだけのリソースを割り当てるかを決める必要がある。いったん構成の評価を始めたら、それを *trial* と呼ぶ。これらの決定を `HPOSearcher` と `HPOScheduler` の 2 つのクラスに対応付ける。さらに、最適化プロセスを実行する `HPOTuner` クラスも提供する。

この scheduler と searcher の概念は、Syne Tune (Salinas *et al.*, 2022)、Ray Tune (Liaw *et al.*, 2018)、Optuna (Akiba *et al.*, 2019) などの一般的な HPO ライブラリにも実装されている。

```
import time
from d2l import torch as d2l
from scipy import stats
```

```
config_space = {
    "learning_rate": stats.loguniform(1e-2, 1),
    "batch_size": stats.randint(32, 256),
}
initial_config = {
    "learning_rate": 0.1,
    "batch_size": 128,
}
```


```
config_space = {
    "learning_rate": stats.loguniform(1e-2, 1),
    "batch_size": stats.randint(32, 256),
}
initial_config = {
    "learning_rate": 0.1,
    "batch_size": 128,
}
```

```
config_space = {
    "learning_rate": stats.loguniform(1e-2, 1),
    "batch_size": stats.randint(32, 256),
}
initial_config = {
    "learning_rate": 0.1,
    "batch_size": 128,
}
```

22.2.1 Searcher

以下では searcher の基底クラスを定義する。これは sample_configuration 関数を通じて新しい候補構成を提供する。この関数を実装する簡単な方法は、Section 22.1 でランダムサーチを行ったときのように、構成を一様ランダムにサンプリングすることである。ベイズ最適化のようなより高度なアルゴリズムでは、過去の trial の性能に基づいてこれらの決定を行う。その結果、時間とともにより有望な候補をサンプリングできるようになる。過去の trial の履歴を更新するために update 関数を追加し、これをサンプリング分布の改善に利用できるようにする。

```
class HPOSearcher(d2l.HyperParameters):  #@save
    def sample_configuration() -> dict:
        raise NotImplementedError

    def update(self, config: dict, error: float, additional_info=None):
        pass
```

```
searcher = RandomSearcher(config_space, initial_config=initial_config)
scheduler = BasicScheduler(searcher=searcher)
tuner = HPOTuner(scheduler=scheduler, objective=hpo_objective_lenet)
tuner.run(number_of_trials=5)
```

```
searcher = RandomSearcher(config_space, initial_config=initial_config)
scheduler = BasicScheduler(searcher=searcher)
tuner = HPOTuner(scheduler=scheduler, objective=hpo_objective_lenet)
tuner.run(number_of_trials=5)
```

```
searcher = RandomSearcher(config_space, initial_config=initial_config)
scheduler = BasicScheduler(searcher=searcher)
tuner = HPOTuner(scheduler=scheduler, objective=hpo_objective_lenet)
tuner.run(number_of_trials=5)
```

次のコードは、前節でのランダムサーチ最適化器をこの API で実装する方法を示している。少し拡張して、最初に評価する構成を initial_config で指定できるようにし、それ以降はランダムにサンプリングする。

```
class RandomSearcher(HPOSearcher):  #@save
    def __init__(self, config_space: dict, initial_config=None):
        self.save_hyperparameters()

    def sample_configuration(self) -> dict:
        if self.initial_config is not None:
            result = self.initial_config
            self.initial_config = None
        else:
            result = {
                name: domain.rvs()
                for name, domain in self.config_space.items()
```

(continues on next page)

(continued from previous page)

```
            }
        return result
```

```
board = d2l.ProgressBoard(xlabel="time", ylabel="error")
for time_stamp, error in zip(
    tuner.cumulative_runtime, tuner.incumbent_trajectory
):
    board.draw(time_stamp, error, "random search", every_n=1)
```


```
board = d2l.ProgressBoard(xlabel="time", ylabel="error")
for time_stamp, error in zip(
    tuner.cumulative_runtime, tuner.incumbent_trajectory
):
    board.draw(time_stamp, error, "random search", every_n=1)
```

```
board = d2l.ProgressBoard(xlabel="time", ylabel="error")
for time_stamp, error in zip(
    tuner.cumulative_runtime, tuner.incumbent_trajectory
):
    board.draw(time_stamp, error, "random search", every_n=1)
```


22.2.2 Scheduler

新しい trial のための構成をサンプリングするだけでなく、trial をいつ、どれだけ長く実行するかも決める必要がある。実際には、これらすべての決定は `HPOScheduler` によって行われ、`HPOSearcher` に新しい構成の選択を委ねる。`suggest` メソッドは、学習のための何らかのリソースが利用可能になるたびに呼び出される。searcher の `sample_configuration` を呼ぶだけでなく、`max_epochs`（つまり、モデルをどれだけ長く学習させるか）などのパラメータも決めることがある。`update` メソッドは、trial が新しい観測値を返すたびに呼び出される。

```
class HPOScheduler(d2l.HyperParameters):  #@save
    def suggest(self) -> dict:
        raise NotImplementedError

    def update(self, config: dict, error: float, info=None):
        raise NotImplementedError
```

ランダムサーチを実装する場合も、他の HPO アルゴリズムを実装する場合も、新しいリソースが利用可能になるたびに新しい構成をスケジュールする基本的な scheduler だけで十分である。

```
class BasicScheduler(HPOScheduler):  #@save
    def __init__(self, searcher: HPOSearcher):
        self.save_hyperparameters()
```

(continues on next page)

(continued from previous page)

```
    def suggest(self) -> dict:
        return self.searcher.sample_configuration()

    def update(self, config: dict, error: float, info=None):
        self.searcher.update(config, error, additional_info=info)
```

22.2.3 Tuner

最後に、scheduler/searcher を実行し、結果をいくつか管理するコンポーネントが必要である。以下のコードは、HPO trial を逐次的に実行し、1 つの学習ジョブを順番に評価するもので、基本例として使える。後ほど、よりスケーラブルな分散 HPO のケースでは *Syne Tune* を使う。

```
class HPOTuner(d2l.HyperParameters):  #@save
    def __init__(self, scheduler: HPOScheduler, objective: callable):
        self.save_hyperparameters()
        # Bookeeping results for plotting
        self.incumbent = None
        self.incumbent_error = None
        self.incumbent_trajectory = []
        self.cumulative_runtime = []
        self.current_runtime = 0
        self.records = []

    def run(self, number_of_trials):
        for i in range(number_of_trials):
            start_time = time.time()
            config = self.scheduler.suggest()
            print(f"Trial {i}: config = {config}")
            error = self.objective(**config)
            error = float(d2l.numpy(error.cpu()))
            self.scheduler.update(config, error)
            runtime = time.time() - start_time
            self.bookkeeping(config, error, runtime)
            print(f"    error = {error}, runtime = {runtime}")
```

22.2.4 HPO アルゴリズムの性能管理

どの HPO アルゴリズムでも、私たちが主に関心を持つのは、与えられた壁時計時間における最良性能の構成（*incumbent* と呼ぶ）と、その検証誤差である。そのため、各反復ごとの `runtime` を追跡する。これには、評価を実行する時間（`objective` の呼び出し）と、意思決定を行う時間（`scheduler.suggest` の呼び出し）の両方が含まれる。以下では、`cumulative_runtime` と `incumbent_trajectory` をプロットして、`scheduler`（および `searcher`）で定義される HPO アルゴリズムの *any-time performance* を可視化する。これにより、最適化器が見つけた構成がどれだけ良いかだけでなく、それをどれだけ速く見つけられるかも定量化できる。

```
@d2l.add_to_class(HPOTuner)  #@save
def bookkeeping(self, config: dict, error: float, runtime: float):
    self.records.append({"config": config, "error": error, "runtime": runtime})
    # Check if the last hyperparameter configuration performs better
    # than the incumbent
    if self.incumbent is None or self.incumbent_error > error:
        self.incumbent = config
        self.incumbent_error = error
    # Add current best observed performance to the optimization trajectory
    self.incumbent_trajectory.append(self.incumbent_error)
    # Update runtime
    self.current_runtime += runtime
    self.cumulative_runtime.append(self.current_runtime)
```

22.2.5 例: 畳み込みニューラルネットワークのハイパーパラメータ最適化

ここでは、新しく実装したランダムサーチを使って、Section 10.6 の LeNet 畳み込みニューラルネットワークの バッチサイズと 学習率を最適化する。まず、目的関数を定義する。ここでも検証誤差を用いる。

```
def hpo_objective_lenet(learning_rate, batch_size, max_epochs=10):  #@save
    model = d2l.LeNet(lr=learning_rate, num_classes=10)
    trainer = d2l.HPOTrainer(max_epochs=max_epochs, num_gpus=1)
    data = d2l.FashionMNIST(batch_size=batch_size)
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
    trainer.fit(model=model, data=data)
    validation_error = trainer.validation_error()
    return validation_error
```

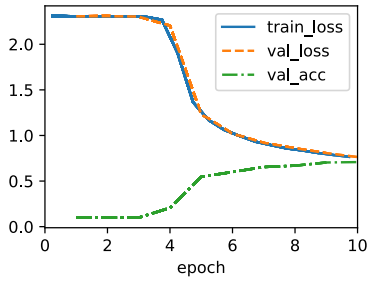
構成空間も定義する必要がある。さらに、最初に評価する構成は Section 10.6 で使ったデフォルト設定とする。

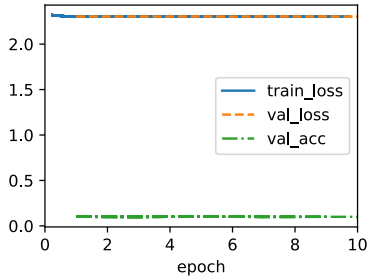
```
config_space = {
    "learning_rate": stats.loguniform(1e-2, 1),
    "batch_size": stats.randint(32, 256),
}
initial_config = {
    "learning_rate": 0.1,
    "batch_size": 128,
}
```

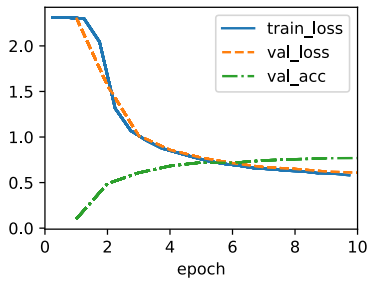
これでランダムサーチを開始できる。

```
searcher = RandomSearcher(config_space, initial_config=initial_config)
scheduler = BasicScheduler(searcher=searcher)
tuner = HPOTuner(scheduler=scheduler, objective=hpo_objective_lenet)
tuner.run(number_of_trials=5)
```

```
    error = 0.42863988876342773, runtime = 69.36778783798218
```





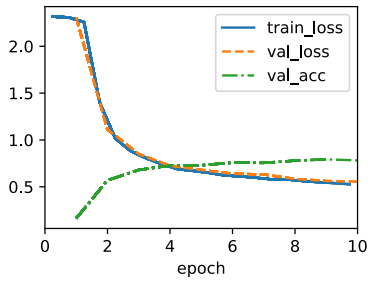


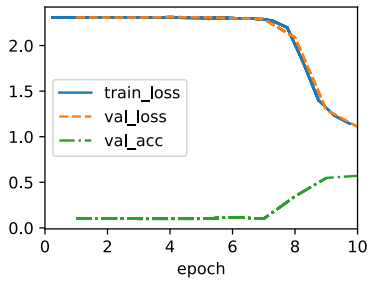
以下では、ランダムサーチの any-time performance を得るために、incumbent の最適化軌跡をプロットする。

```
board = d2l.ProgressBoard(xlabel="time", ylabel="error")
for time_stamp, error in zip(
    tuner.cumulative_runtime, tuner.incumbent_trajectory
):
    board.draw(time_stamp, error, "random search", every_n=1)
```

22.2.6 HPO アルゴリズムの比較

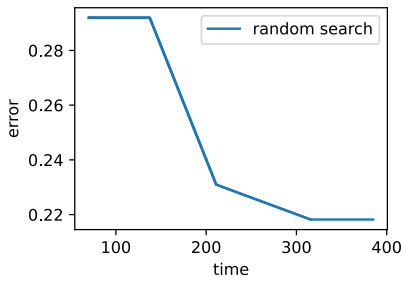
学習アルゴリズムやモデルアーキテクチャと同様に、異なる HPO アルゴリズムをどのように比較するのが最善かを理解することは重要である。各 HPO 実





行は、2 つの主要なランダム性の源に依存する。1 つは、ランダムな重み初期化やミニバッチの順序付けなど、学習過程に由来するランダムな効果である。もう 1 つは、ランダムサーチにおけるランダムサンプリングのような、HPO アルゴリズム自体に内在するランダム性である。したがって、異なるアルゴリズムを比較する際には、各実験を複数回実行し、乱数生成器の異なるシードに基づく複数回の反復全体で、平均や中央値などの統計量を報告することが重要である。

これを示すために、フィードフォワードニューラルネットワークのハイパーパラメータ調整において、ランダムサーチ（Section 22.1.2 を参照）とベイズ最適化 (Snoek *et al.*, 2012) を比較する。各アルゴリズムは、異なる乱数シードで 50 回評価された。実線はこの 50 回の反復にわたる incumbent の平均性能を示し、破線は標準偏差を示す。ランダムサーチとベイズ最適化はおよそ 1000 秒まではほぼ同程度の性能であるが、ベイズ最適化は過去の観測を利用してより良い構成を特定できるため、その後はすぐにランダムサーチを上回ることがわかる。



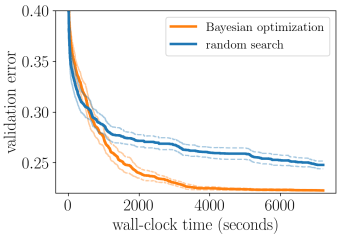


Fig. 22.2.1 2 つのアルゴリズム A と B を比較するための any-time performance プロットの例。

22.2.7 まとめ

この節では、この章で扱うさまざまな HPO アルゴリズムを実装するための、シンプルでありながら柔軟なインターフェースを示した。類似のインターフェースは、一般的なオープンソースの HPO フレームワークにも見られる。また、HPO アルゴリズムをどのように比較できるか、そして注意すべき潜在的な落とし穴についても見た。

22.2.8 演習

1. この演習の目的は、少し難しめの HPO 問題の目的関数を実装し、より現実的な実験を行うことである。Section 8.6 で実装した 2 層隠れ層 MLP `DropoutMLP` を使う。
   1. 目的関数をコード化せよ。これはモデルのすべてのハイパーパラメータと `batch_size` に依存する必要がある。`max_epochs=50` を使用せよ。ここでは GPU は役に立たないため、`num_gpus=0` とする。ヒント: `hpo_objective_lenet` を修正せよ。
   2. 妥当な探索空間を選べ。`num_hiddens_1`、`num_hiddens_2` は $[8, 1024]$ の整数、dropout 値は $[0, 0.95]$、`batch_size` は $[16, 384]$ とする。`scipy.stats` の適切な分布を使って `config_space` のコードを示せ。
   3. この例で `number_of_trials=20` としてランダムサーチを実行し、結果をプロットせよ。まず Section 8.6 のデフォルト構成、すなわち `initial_config = {'num_hiddens_1': 256, 'num_hiddens_2': 256, 'dropout_1': 0.5, 'dropout_2': 0.5, 'lr': 0.1, 'batch_size': 256}` を最初に評価することを忘れないこと。
2. この演習では、過去のデータに基づいて意思決定を行う新しい searcher（`HPOSearcher` のサブクラス）を実装する。これは `probab_local`、`num_init_random` というパラメータに依存する。`sample_configuration` メソッドは次のように動作する。最初の `num_init_random` 回の呼び出しでは、`RandomSearcher.sample_configuration` と同じことを行う。それ以外では、確率 `1 - probab_local` で `RandomSearcher.sample_configuration` と同じことを行う。それ以外では、これまでで最小の検証誤差を達成した構成を選び、そのハイパーパラメータの 1 つをランダムに選んで、その値を `RandomSearcher.sample_configuration`

と同様にランダムにサンプリングするが、他の値はそのままにする。この 1 つのハイパーパラメータだけが異なり、それ以外はこれまでの最良構成と同一である構成を返せ。

1. この新しい `LocalSearcher` をコード化せよ。ヒント: searcher の構築時には引数として `config_space` が必要である。`RandomSearcher` 型のメンバーを使っても構わない。また、`update` メソッドも実装する必要がある。
2. 前の演習の実験を、`RandomSearcher` の代わりにこの新しい searcher を使って再実行せよ。`probab_local`、`num_init_random` のさまざまな値を試せ。ただし、異なる HPO 手法を適切に比較するには、実験を複数回繰り返し、できれば複数のベンチマークタスクを考慮する必要があることに注意せよ。

22.3 非同期ランダムサーチ

前の Section 22.2 で見たように、ハイパーパラメータ構成の評価にはコストがかかるため、ランダムサーチが良いハイパーパラメータ構成を返すまでに数時間、あるいは数日待たなければならないことがある。実際には、同じマシン上の複数 GPU や、単一 GPU を持つ複数マシンといった資源プールを利用できることがよくある。ここで問題になるのは、ランダムサーチをどのように効率よく分散するかである。

一般に、同期型と非同期型の並列ハイパーパラメータ最適化を区別する（Fig. 22.3.1 を参照）。同期型では、同時に実行中のすべての試行が終了するまで待ってから、次のバッチを開始する。深層ニューラルネットワークのフィルタ数や層数のようなハイパーパラメータを含む構成空間を考えてみよう。より多くのフィルタ層を含むハイパーパラメータ構成は、当然ながら完了までにより長い時間を要し、同じバッチ内の他のすべての試行は、最適化プロセスを続ける前に同期点（Fig. 22.3.1 の灰色部分）で待たなければならない。

非同期型では、資源が利用可能になり次第、すぐに新しい試行をスケジュールする。これにより、同期のオーバーヘッドを避けられるため、資源を最適に活用できる。ランダムサーチでは、新しいハイパーパラメータ構成はそれぞれ他のすべてから独立に選ばれ、特に過去の評価結果を利用しない。つまり、ランダムサーチは非同期に容易に並列化できる。これは、過去の観測に基づいて意思決定を行う、より高度な手法では自明ではない（Section 22.5 を参照）。逐次設定よりも多くの資源にアクセスできる必要はあるが、非同期ランダムサーチは線形の高速化を示し、K 個の試行を並列実行できれば、ある性能に到達するまでの時間は K 倍速くなる。

このノートブックでは、同一マシン上の複数の python プロセスで試行を実行する非同期ランダムサーチを見ていく。分散ジョブのスケジューリングと実行をゼロから実装するのは困難である。そこで、非同期 HPO のためのシンプルなイ

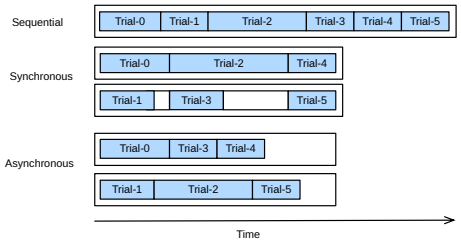


Fig. 22.3.1 ハイパーパラメータ最適化プロセスを同期的または非同期的に分散する。逐次設定と比べて、総計算量を一定に保ったまま、全体の壁時計時間を短縮できる。同期スケジューリングでは、遅い試行がある場合にワーカーが遊休状態になることがある。

ンターフェースを提供する *Syne Tune* (Salinas *et al.*, 2022) を使用する。Syne Tune はさまざまな実行バックエンドで動作するよう設計されており、分散 HPO についてさらに学びたい読者は、そのシンプルな API をぜひ調べてみよ。

```
from d2l import torch as d2l
import logging
logging.basicConfig(level=logging.INFO)
from syne_tune.config_space import loguniform, randint
from syne_tune.backend.python_backend import PythonBackend
from syne_tune.optimizer.baselines import RandomSearch
from syne_tune import Tuner, StoppingCriterion
from syne_tune.experiments import load_experiment
```

```
INFO:root:SageMakerBackend is not imported since dependencies are missing. You␣
↪can install them with
   pip install 'syne-tune[extra]'
AWS dependencies are not imported since dependencies are missing. You can␣
↪install them with
   pip install 'syne-tune[aws]'
or (for everything)
   pip install 'syne-tune[extra]'
AWS dependencies are not imported since dependencies are missing. You can␣
↪install them with
   pip install 'syne-tune[aws]'
or (for everything)
   pip install 'syne-tune[extra]'
INFO:root:Ray Tune schedulers and searchers are not imported since␣
↪dependencies are missing. You can install them with
   pip install 'syne-tune[raytune]'
or (for everything)
   pip install 'syne-tune[extra]'
```

22.3.1 目的関数

まず、新しい目的関数を定義し、`report` コールバックを通じて Syne Tune に性能を返すようにする。

```
def hpo_objective_lenet_synetune(learning_rate, batch_size, max_epochs):
    from d2l import torch as d2l
    from syne_tune import Reporter

    model = d2l.LeNet(lr=learning_rate, num_classes=10)
    trainer = d2l.HPOTrainer(max_epochs=1, num_gpus=1)
    data = d2l.FashionMNIST(batch_size=batch_size)
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
    report = Reporter()
    for epoch in range(1, max_epochs + 1):
        if epoch == 1:
            # Trainer の状態を初期化する
            trainer.fit(model=model, data=data)
        else:
            trainer.fit_epoch()
        validation_error = d2l.numpy(trainer.validation_error().cpu())
        report(epoch=epoch, validation_error=float(validation_error))
```

Syne Tune の PythonBackend では、依存関係を関数定義の内部で import する必要があることに注意せよ。

22.3.2 非同期スケジューラ

まず、同時に試行を評価するワーカー数を定義する。また、ランダムサーチをどれくらいの時間実行したいかを、総壁時計時間の上限として指定する必要がある。

```
n_workers = 2  # 利用可能な GPU 数以下である必要がある

max_wallclock_time = 12 * 60  # 12 分
```

次に、最適化したい指標と、その指標を最小化するか最大化するかを指定する。つまり、metric は report コールバックに渡す引数名に対応していなければならない。

```
mode = "min"
metric = "validation_error"
```

前の例で使った構成空間を用いる。Syne Tune では、この辞書を使って学習スクリプトに定数属性を渡すこともできる。ここでは max_epochs を渡すためにこの機能を利用する。さらに、最初に評価する構成を initial_config で指定する。

```
config_space = {
    "learning_rate": loguniform(1e-2, 1),
    "batch_size": randint(32, 256),
    "max_epochs": 10,
}
initial_config = {
```

(continues on next page)

(continued from previous page)

```
    "learning_rate": 0.1,
    "batch_size": 128,
}
```

次に、ジョブ実行のバックエンドを指定する必要がある。ここでは、並列ジョブがサブプロセスとして実行されるローカルマシン上での分散のみを考える。しかし、大規模な HPO では、各試行が 1 つのインスタンス全体を消費するクラスタ環境やクラウド環境でも実行できる。

```
trial_backend = PythonBackend(
    tune_function=hpo_objective_lenet_synetune,
    config_space=config_space,
)
```

これで、非同期ランダムサーチのスケジューラを作成できる。その動作は Section 22.2 の `BasicScheduler` に似ている。

```
scheduler = RandomSearch(
    config_space,
    metric=metric,
    mode=mode,
    points_to_evaluate=[initial_config],
)
```

```
INFO:syne_tune.optimizer.schedulers.fifo:max_resource_level = 10, as inferred␣
↪from config_space
INFO:syne_tune.optimizer.schedulers.fifo:Master random_seed = 952911033
```

Syne Tune には `Tuner` もあり、主要な実験ループと管理処理が集中化され、スケジューラとバックエンドのやり取りが仲介される。

```
stop_criterion = StoppingCriterion(max_wallclock_time=max_wallclock_time)

tuner = Tuner(
    trial_backend=trial_backend,
    scheduler=scheduler,
    stop_criterion=stop_criterion,
    n_workers=n_workers,
    print_update_interval=int(max_wallclock_time * 0.6),
)
```

分散 HPO 実験を実行してみよう。停止条件に従って、およそ 12 分間実行される。

```
tuner.run()
```

```
INFO:syne_tune.tuner:results of trials will be saved on /home/ci/syne-tune/
↪python-entrypoint-2023-08-18-07-32-49-046
INFO:root:Detected 4 GPUs
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.1 --batch_size 128 --max_epochs 10 --tune_function_root␣
↪/home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/tune_function --
↪tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_checkpoint_dir /
↪home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/0/checkpoints
INFO:syne_tune.tuner:(trial 0) - scheduled config {'learning_rate': 0.1,
↪'batch_size': 128, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.023130676592269524 --batch_size 126 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪1/checkpoints
INFO:syne_tune.tuner:(trial 1) - scheduled config {'learning_rate': 0.
↪023130676592269524, 'batch_size': 126, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 0 completed.
INFO:syne_tune.tuner:Trial trial_id 1 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.312685188079284 --batch_size 86 --max_epochs 10 --tune_
↪function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪2/checkpoints
INFO:syne_tune.tuner:(trial 2) - scheduled config {'learning_rate': 0.
↪312685188079284, 'batch_size': 86, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.060054251769848306 --batch_size 109 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪3/checkpoints
INFO:syne_tune.tuner:(trial 3) - scheduled config {'learning_rate': 0.
↪060054251769848306, 'batch_size': 109, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 3 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.055145478348008786 --batch_size 62 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪4/checkpoints
INFO:syne_tune.tuner:(trial 4) - scheduled config {'learning_rate': 0.
↪055145478348008786, 'batch_size': 62, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 2 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.3071404894525763 --batch_size 175 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
```

(continues on next page)

(continued from previous page)

```
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪5/checkpoints
INFO:syne_tune.tuner:(trial 5) - scheduled config {'learning_rate': 0.
↪3071404894525763, 'batch_size': 175, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 5 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.3105275803794171 --batch_size 57 --max_epochs 10 --tune_
↪function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪6/checkpoints
INFO:syne_tune.tuner:(trial 6) - scheduled config {'learning_rate': 0.
↪3105275803794171, 'batch_size': 57, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 4 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.014311777503766778 --batch_size 251 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪7/checkpoints
INFO:syne_tune.tuner:(trial 7) - scheduled config {'learning_rate': 0.
↪014311777503766778, 'batch_size': 251, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 7 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.08111483940636906 --batch_size 112 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪8/checkpoints
INFO:syne_tune.tuner:(trial 8) - scheduled config {'learning_rate': 0.
↪08111483940636906, 'batch_size': 112, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 6 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.026212700431916515 --batch_size 188 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪9/checkpoints
INFO:syne_tune.tuner:(trial 9) - scheduled config {'learning_rate': 0.
↪026212700431916515, 'batch_size': 188, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 8 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.1325223401116132 --batch_size 37 --max_epochs 10 --tune_
↪function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪10/checkpoints
INFO:syne_tune.tuner:(trial 10) - scheduled config {'learning_rate': 0.
↪1325223401116132, 'batch_size': 37, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 9 completed.
```

(continues on next page)

(continued from previous page)

```
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.2797904076893328 --batch_size 251 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪11/checkpoints
INFO:syne_tune.tuner:(trial 11) - scheduled config {'learning_rate': 0.
↪2797904076893328, 'batch_size': 251, 'max_epochs': 10}
INFO:syne_tune.tuner:tuning status (last metric is reported)
 trial_id     status  iter  learning_rate  batch_size  max_epochs  epoch ␣
↪validation_error  worker-time
        0  Completed    10       0.100000         128          10   10.0      ␣
↪   0.276305    70.635542
        1  Completed    10       0.023131         126          10   10.0      ␣
↪   0.900276    70.564567
        2  Completed    10       0.312685          86          10   10.0      ␣
↪   0.177773    76.995033
        3  Completed    10       0.060054         109          10   10.0      ␣
↪   0.336481    72.600473
        4  Completed    10       0.055145          62          10   10.0      ␣
↪   0.260045    90.789169
        5  Completed    10       0.307140         175          10   10.0      ␣
↪   0.234286    73.124661
        6  Completed    10       0.310528          57          10   10.0      ␣
↪   0.150503    94.374417
        7  Completed    10       0.014312         251          10   10.0      ␣
↪   0.900077    73.351970
        8  Completed    10       0.081115         112          10   10.0      ␣
↪   0.274901    77.767942
        9  Completed    10       0.026213         188          10   10.0      ␣
↪   0.899002    73.504148
       10 InProgress     1       0.132522          37          10    1.0      ␣
↪   0.417084    11.319020
       11 InProgress     0       0.279790         251          10      -      ␣
↪          -           -
2 trials running, 10 finished (10 until the end), 436.62s wallclock-time

INFO:syne_tune.tuner:Trial trial_id 11 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.02754593050837886 --batch_size 183 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪12/checkpoints
INFO:syne_tune.tuner:(trial 12) - scheduled config {'learning_rate': 0.
↪02754593050837886, 'batch_size': 183, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 10 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.01099464105506646 --batch_size 105 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
```

(continues on next page)

(continued from previous page)

```
↪13/checkpoints
INFO:syne_tune.tuner:(trial 13) - scheduled config {'learning_rate': 0.
↪01099464105506646, 'batch_size': 105, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 12 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.06457279133966501 --batch_size 252 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪14/checkpoints
INFO:syne_tune.tuner:(trial 14) - scheduled config {'learning_rate': 0.
↪06457279133966501, 'batch_size': 252, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 13 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.15504555979876097 --batch_size 32 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪15/checkpoints
INFO:syne_tune.tuner:(trial 15) - scheduled config {'learning_rate': 0.
↪15504555979876097, 'batch_size': 32, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 14 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.7645122914150699 --batch_size 246 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪16/checkpoints
INFO:syne_tune.tuner:(trial 16) - scheduled config {'learning_rate': 0.
↪7645122914150699, 'batch_size': 246, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 16 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.10143987295366007 --batch_size 247 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-
↪046/tune_function --tune_function_hash 6d601acbbb4647b6d65d87860369b1db --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046/
↪17/checkpoints
INFO:syne_tune.tuner:(trial 17) - scheduled config {'learning_rate': 0.
↪10143987295366007, 'batch_size': 247, 'max_epochs': 10}
INFO:syne_tune.stopping_criterion:reaching max wallclock time (720), stopping␣
↪there.
INFO:syne_tune.tuner:Stopping trials that may still be running.
INFO:syne_tune.tuner:Tuning finished, results of trials can be found on /home/
↪ci/syne-tune/python-entrypoint-2023-08-18-07-32-49-046
--------------------
Resource summary (last result is reported):
 trial_id     status  iter  learning_rate  batch_size  max_epochs  epoch ␣
↪validation_error  worker-time
        0  Completed    10       0.100000         128          10     10      ␣
↪   0.276305    70.635542
        1  Completed    10       0.023131         126          10     10      ␣
```

(continues on next page)

(continued from previous page)

```
↪   0.900276    70.564567
       2  Completed    10       0.312685          86          10     10       ␣
↪   0.177773    76.995033
       3  Completed    10       0.060054         109          10     10       ␣
↪   0.336481    72.600473
       4  Completed    10       0.055145          62          10     10       ␣
↪   0.260045    90.789169
       5  Completed    10       0.307140         175          10     10       ␣
↪   0.234286    73.124661
       6  Completed    10       0.310528          57          10     10       ␣
↪   0.150503    94.374417
       7  Completed    10       0.014312         251          10     10       ␣
↪   0.900077    73.351970
       8  Completed    10       0.081115         112          10     10       ␣
↪   0.274901    77.767942
       9  Completed    10       0.026213         188          10     10       ␣
↪   0.899002    73.504148
      10  Completed    10       0.132522          37          10     10       ␣
↪   0.161034   119.344017
      11  Completed    10       0.279790         251          10     10       ␣
↪   0.445979    68.693048
      12  Completed    10       0.027546         183          10     10       ␣
↪   0.803873    67.717216
      13  Completed    10       0.010995         105          10     10       ␣
↪   0.859286    68.206863
      14  Completed    10       0.064573         252          10     10       ␣
↪   0.551479    58.861913
      15 InProgress     5       0.155046          32          10      5       ␣
↪   0.193590    81.370336
      16  Completed    10       0.764512         246          10     10       ␣
↪   0.178272    57.425193
      17 InProgress     2       0.101440         247          10      2       ␣
↪   0.825418    10.832541
2 trials running, 16 finished (16 until the end), 722.85s wallclock-time

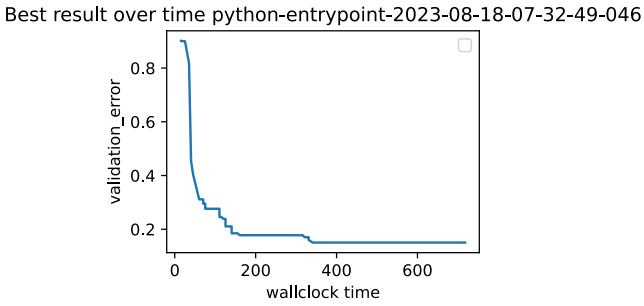
validation_error: best 0.15050286054611206 for trial-id 6
--------------------
```

評価したすべてのハイパーパラメータ構成のログは、後で分析できるように保存される。チューニングジョブの実行中いつでも、これまでに得られた結果を簡単に取得し、incumbent の軌跡を描画できる。

```
d2l.set_figsize()
tuning_experiment = load_experiment(tuner.name)
tuning_experiment.plot()
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note␣
↪that artists whose label start with an underscore are ignored when legend()␣
↪is called with no argument.
```

22.3.3 非同期最適化プロセスの可視化



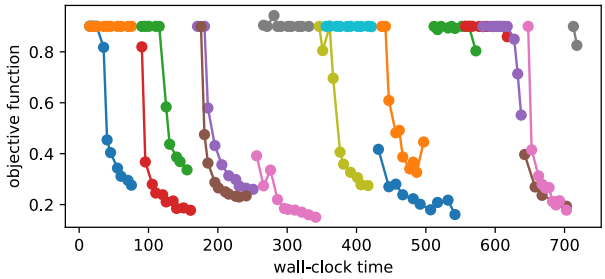
以下では、各試行の学習曲線（プロット内の各色が 1 つの試行を表す）が非同期最適化プロセスの間にどのように変化するかを可視化する。任意の時点で、同時に実行されている試行数はワーカー数と同じである。ある試行が終了すると、他の試行の終了を待たずに、すぐ次の試行を開始する。非同期スケジューリングでは、ワーカーの遊休時間を最小限に抑えられる。

```
d2l.set_figsize([6, 2.5])
results = tuning_experiment.results

for trial_id in results.trial_id.unique():
    df = results[results["trial_id"] == trial_id]
    d2l.plt.plot(
        df["st_tuner_time"],
        df["validation_error"],
        marker="o"
    )

d2l.plt.xlabel("wall-clock time")
d2l.plt.ylabel("objective function")
```

```
Text(0, 0.5, 'objective function')
```



22.3.4 まとめ

試行を並列資源に分散することで、ランダムサーチの待ち時間を大幅に短縮できる。一般に、同期スケジューリングと非同期スケジューリングを区別する。

同期スケジューリングでは、前のバッチが終了してから新しいハイパーパラメータ構成のバッチをサンプリングする。遅い試行、つまり他の試行より完了に時間がかかる試行があると、ワーカーは同期点で待たなければならない。非同期スケジューリングでは、資源が利用可能になり次第、新しいハイパーパラメータ構成を評価するため、任意の時点で全ワーカーが稼働していることが保証される。ランダムサーチは非同期に容易に分散でき、実際のアルゴリズム自体を変更する必要はないが、他の手法では追加の修正が必要になる。

22.3.5 演習

1. Section 8.6 で実装され、Section 22.2 の演習 1 で使用した `DropoutMLP` モデルを考える。

   1. Syne Tune で使う目的関数 `hpo_objective_dropoutmlp_synetune` を実装しなさい。関数が各エポック後に検証誤差を報告することを確認しなさい。

   2. Section 22.2 の演習 1 の設定を用いて、ランダムサーチとベイズ最適化を比較しなさい。SageMaker を使う場合は、Syne Tune のベンチマーク機能を利用して実験を並列実行してよい。ヒント: ベイズ最適化は `syne_tune.optimizer.baselines.BayesianOptimization` として提供されている。

   3. この演習では、少なくとも 4 つの CPU コアを持つインスタンスで実行する必要がある。上で使った手法のうち 1 つ（ランダムサーチ、ベイズ最適化）について、`n_workers=1`、`n_workers=2`、`n_workers=4` で実験を行い、結果（incumbent の軌跡）を比較しなさい。少なくともランダムサーチでは、ワーカー数に対して線形スケーリングが観測されるはずである。ヒント: 安定した結果を得るには、それぞれ複数回繰り返して平均を取る必要があるかもしれない。

127 

2. 発展. この演習の目標は、Syne Tune に新しいスケジューラを実装することである。

128 

   1. d2lbook[127] と syne-tune[128] の両方のソースを含む仮想環境を作成しなさい。

129 

   2. Section 22.2 の演習 2 での `LocalSearcher` を、Syne Tune の新しいサーチャーとして実装しなさい。ヒント: このチュートリアル[129] を読んでよ。あるいは、この 例[130] に従ってもよいである。

130 

   3. 新しく実装した `LocalSearcher` と `DropoutMLP` ベンチマーク上の `RandomSearch` を比較しなさい。

22.4 マルチフィデリティ・ハイパーパラメータ最適化

ニューラルネットワークの学習は、中規模のデータセットであっても高コストになり得る。構成空間（Section 22.1.1）によっては、ハイパーパラメータ最適化では、性能の良いハイパーパラメータ構成を見つけるために数十回から数百回の関数評価が必要になる。Section 22.3 で見たように、並列リソースを活用することで HPO の全体的な壁時計時間を大幅に短縮できるが、必要な総計算量は減らない。

この節では、ハイパーパラメータ構成の評価をどのように高速化できるかを示す。ランダムサーチのような手法では、各ハイパーパラメータ評価に同じ量のリソース（たとえばエポック数や学習データ点数）を割り当てる。Fig. 22.4.1 は、異なるハイパーパラメータ構成で学習したニューラルネットワーク群の学習曲線を示している。数エポック後には、性能の良い構成と不十分な構成を視覚的に区別できるようになる。しかし、学習曲線にはノイズがあるため、最良の構成を特定するには依然として 100 エポック分の完全な評価が必要かもしれない。

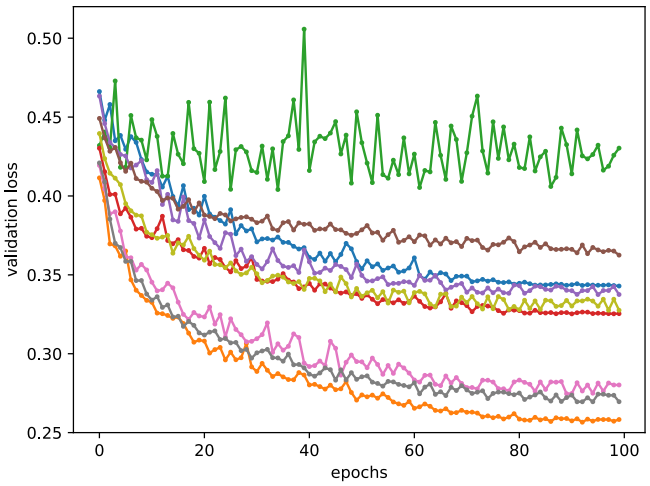


Fig. 22.4.1 ランダムなハイパーパラメータ構成の学習曲線

マルチフィデリティ・ハイパーパラメータ最適化では、有望な構成により多くのリソースを割り当て、性能の悪い構成の評価は早期に打ち切りる。これにより、同じ総リソース量でより多くの構成を試せるため、最適化プロセスが高速化される。

より形式的には、Section 22.1.1 における定義を拡張し、目的関数 $f(\mathbf{x}, r)$ に追加

の入力 $r \in [r_{\min}, r_{max}]$ を導入する。これは、構成 $\mathbf{x}$ の評価に対してどれだけのリソースを費やすかを指定する。ここでは、誤差 $f(\mathbf{x}, r)$ は r とともに減少し、計算コスト $c(\mathbf{x}, r)$ は増加すると仮定する。通常、r はニューラルネットワークの学習エポック数を表すが、学習サブセットのサイズや交差検証の分割数でもかまわない。

```
from d2l import torch as d2l
import numpy as np
from scipy import stats
from collections import defaultdict
d2l.set_figsize()
```

```
class SuccessiveHalvingScheduler(d2l.HPOScheduler):  #@save
    def __init__(self, searcher, eta, r_min, r_max, prefact=1):
        self.save_hyperparameters()
        # Compute K, which is later used to determine the number of␣
↪configurations
        self.K = int(np.log(r_max / r_min) / np.log(eta))
        # Define the rungs
        self.rung_levels = [r_min * eta ** k for k in range(self.K + 1)]
        if r_max not in self.rung_levels:
            # The final rung should be r_max
            self.rung_levels.append(r_max)
            self.K += 1
        # Bookkeeping
        self.observed_error_at_rungs = defaultdict(list)
        self.all_observed_error_at_rungs = defaultdict(list)
        # Our processing queue
        self.queue = []
```

```
class SuccessiveHalvingScheduler(d2l.HPOScheduler):  #@save
    def __init__(self, searcher, eta, r_min, r_max, prefact=1):
        self.save_hyperparameters()
        # Compute K, which is later used to determine the number of␣
↪configurations
        self.K = int(np.log(r_max / r_min) / np.log(eta))
        # Define the rungs
        self.rung_levels = [r_min * eta ** k for k in range(self.K + 1)]
        if r_max not in self.rung_levels:
            # The final rung should be r_max
            self.rung_levels.append(r_max)
            self.K += 1
        # Bookkeeping
        self.observed_error_at_rungs = defaultdict(list)
        self.all_observed_error_at_rungs = defaultdict(list)
        # Our processing queue
        self.queue = []
```

```
class SuccessiveHalvingScheduler(d2l.HPOScheduler):  #@save
    def __init__(self, searcher, eta, r_min, r_max, prefact=1):
        self.save_hyperparameters()
        # Compute K, which is later used to determine the number of␣
```

(continues on next page)

(continued from previous page)

```
↪configurations
        self.K = int(np.log(r_max / r_min) / np.log(eta))
        # Define the rungs
        self.rung_levels = [r_min * eta ** k for k in range(self.K + 1)]
        if r_max not in self.rung_levels:
            # The final rung should be r_max
            self.rung_levels.append(r_max)
            self.K += 1
        # Bookkeeping
        self.observed_error_at_rungs = defaultdict(list)
        self.all_observed_error_at_rungs = defaultdict(list)
        # Our processing queue
        self.queue = []
```

22.4.1 逐次半減法

マルチフィデリティ設定にランダムサーチを適応させる最も簡単な方法の 1 つが、逐次半減法 (Jamieson and Talwalkar, 2016, Karnin *et al.*, 2013) である。基本的な考え方は、たとえば構成空間からランダムにサンプルした N 個の構成から始め、それぞれを $r_{\min}$ エポックだけ学習することである。その後、性能の悪い試行の一部を捨て、残ったものをより長く学習する。このプロセスを繰り返すことで、より少ない試行がより長く実行され、少なくとも 1 つの試行が r_{max} エポックに到達するまで続く。

より形式的には、最小予算 $r_{\min}$（たとえば 1 エポック）、最大予算 r_{max}（たとえば前の例の `max_epochs`）、および半減定数 $\eta \in \{2, 3, \dots\}$ を考える。簡単のために、$r_{max} = r_{\min}\eta^K$、ただし $K \in \mathbb{I}$ と仮定する。このとき初期構成数は $N = \eta^K$ である。段（rung）の集合を $\mathcal{R} = \{r_{\min}, r_{\min}\eta, r_{\min}\eta^2, \dots, r_{max}\}$ と定義する。

逐次半減法の 1 ラウンドは次のように進みる。まず N 個の試行を最初の段 $r_{\min}$ まで実行する。検証誤差を並べ替え、上位 $1/\eta$ の割合（これは η^{K-1} 個の構成に相当する）を残し、それ以外はすべて破棄する。生き残った試行は次の段（$r_{\min}\eta$ エポック）まで学習され、この過程を繰り返す。各段で試行の $1/\eta$ が生き残り、その学習は η 倍大きい予算で継続される。この N の選び方では、最終的に 1 つの試行だけが完全な予算 r_{max} まで学習される。このような逐次半減法の 1 ラウンドが終わると、新しい初期構成集合で次のラウンドを開始し、総予算を使い切るまで繰り返す。

逐次半減法を実装するために、Section 22.2 の `HPOScheduler` 基底クラスを継承し、汎用の `HPOSearcher` オブジェクトが構成をサンプルできるようにする（以下の例では `RandomSearcher` になる）。さらに、ユーザーは最小リソース $r_{\min}$、最大リソース r_{max}、および η を入力として渡す必要がある。スケジューラ内部では、現在の段 r_i でまだ評価が必要な構成のキューを保持する。次の段へ進むたびに、このキューを更新する。

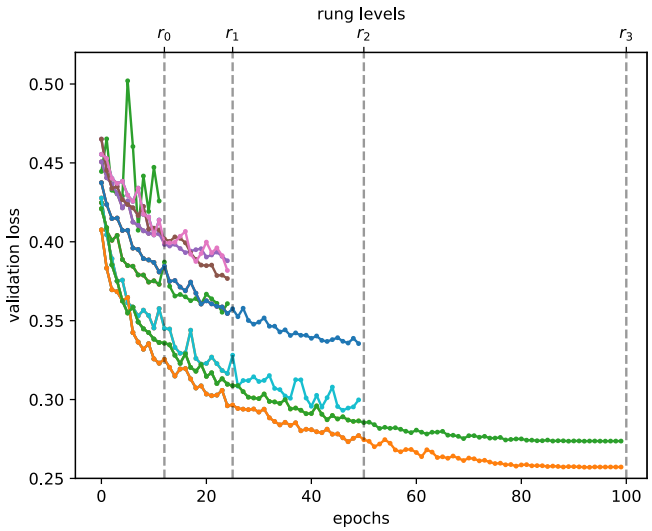


Fig. 22.4.2 ランダムなハイパーパラメータ構成の学習曲線。

```
class SuccessiveHalvingScheduler(d2l.HPOScheduler):  #@save
    def __init__(self, searcher, eta, r_min, r_max, prefact=1):
        self.save_hyperparameters()
        # Compute K, which is later used to determine the number of␣
↪configurations
        self.K = int(np.log(r_max / r_min) / np.log(eta))
        # Define the rungs
        self.rung_levels = [r_min * eta ** k for k in range(self.K + 1)]
        if r_max not in self.rung_levels:
            # The final rung should be r_max
            self.rung_levels.append(r_max)
            self.K += 1
        # Bookkeeping
        self.observed_error_at_rungs = defaultdict(list)
        self.all_observed_error_at_rungs = defaultdict(list)
        # Our processing queue
        self.queue = []
```

```
min_number_of_epochs = 2
max_number_of_epochs = 10
eta = 2
num_gpus=1

config_space = {
    "learning_rate": stats.loguniform(1e-2, 1),
    "batch_size": stats.randint(32, 256),
}
initial_config = {
    "learning_rate": 0.1,
    "batch_size": 128,
}
```

```
min_number_of_epochs = 2
max_number_of_epochs = 10
eta = 2
num_gpus=1

config_space = {
    "learning_rate": stats.loguniform(1e-2, 1),
    "batch_size": stats.randint(32, 256),
}
initial_config = {
    "learning_rate": 0.1,
    "batch_size": 128,
}
```


```
min_number_of_epochs = 2
max_number_of_epochs = 10
eta = 2
num_gpus=1

config_space = {
    "learning_rate": stats.loguniform(1e-2, 1),
    "batch_size": stats.randint(32, 256),
}
initial_config = {
    "learning_rate": 0.1,
    "batch_size": 128,
}
```


最初はキューは空で、そこに $n = \text{prefact} \cdot \eta^K$ 個の構成を入れる。それらはまず最小の段 $r_{\min}$ で評価される。ここで prefact は、別の文脈でこのコードを再利用できるようにするためのものである。この節では、prefact = 1 に固定する。リソースが利用可能になり、`HPOTuner` オブジェクトが `suggest` 関数を呼び出すたびに、キューから 1 つの要素を返す。逐次半減法の 1 ラウンドが完了し、つまり生き残ったすべての構成を最高リソースレベル r_{max} まで評価し終えてキューが空になったら、新しいランダムサンプルの構成集合でプロセス全体を再開する。

```
@d2l.add_to_class(SuccessiveHalvingScheduler)  #@save
def suggest(self):
    if len(self.queue) == 0:
        # Start a new round of successive halving
        # Number of configurations for the first rung:
        n0 = int(self.prefact * self.eta ** self.K)
        for _ in range(n0):
            config = self.searcher.sample_configuration()
            config["max_epochs"] = self.r_min  # Set r = r_min
            self.queue.append(config)
    # Return an element from the queue
    return self.queue.pop()
```

```
searcher = d2l.RandomSearcher(config_space, initial_config=initial_config)
scheduler = SuccessiveHalvingScheduler(
    searcher=searcher,
    eta=eta,
    r_min=min_number_of_epochs,
    r_max=max_number_of_epochs,
)
tuner = d2l.HPOTuner(
    scheduler=scheduler,
    objective=d2l.hpo_objective_lenet,
)
tuner.run(number_of_trials=30)
```

```
searcher = d2l.RandomSearcher(config_space, initial_config=initial_config)
scheduler = SuccessiveHalvingScheduler(
    searcher=searcher,
    eta=eta,
    r_min=min_number_of_epochs,
    r_max=max_number_of_epochs,
)
tuner = d2l.HPOTuner(
    scheduler=scheduler,
    objective=d2l.hpo_objective_lenet,
)
tuner.run(number_of_trials=30)
```

```
searcher = d2l.RandomSearcher(config_space, initial_config=initial_config)
scheduler = SuccessiveHalvingScheduler(
    searcher=searcher,
    eta=eta,
    r_min=min_number_of_epochs,
    r_max=max_number_of_epochs,
)
tuner = d2l.HPOTuner(
    scheduler=scheduler,
    objective=d2l.hpo_objective_lenet,
)
tuner.run(number_of_trials=30)
```

新しいデータ点を収集したら、まず searcher モジュールを更新する。その後、現在の段のすべてのデータ点をすでに収集し終えたかを確認する。もしそうなら、すべての構成を並べ替え、上位 $\frac{1}{\eta}$ の構成をキューに入れる。

```
@d2l.add_to_class(SuccessiveHalvingScheduler)  #@save
def update(self, config: dict, error: float, info=None):
    ri = int(config["max_epochs"])  # Rung r_i
    # Update our searcher, e.g if we use Bayesian optimization later
    self.searcher.update(config, error, additional_info=info)
    self.all_observed_error_at_rungs[ri].append((config, error))
    if ri < self.r_max:
        # Bookkeeping
        self.observed_error_at_rungs[ri].append((config, error))
```

(continues on next page)

(continued from previous page)

```
        # Determine how many configurations should be evaluated on this rung
        ki = self.K - self.rung_levels.index(ri)
        ni = int(self.prefact * self.eta ** ki)
        # If we observed all configuration on this rung r_i, we estimate the
        # top 1 / eta configuration, add them to queue and promote them for
        # the next rung r_{i+1}
        if len(self.observed_error_at_rungs[ri]) >= ni:
            kiplus1 = ki - 1
            niplus1 = int(self.prefact * self.eta ** kiplus1)
            best_performing_configurations = self.get_top_n_configurations(
                rung_level=ri, n=niplus1
            )
            riplus1 = self.rung_levels[self.K - kiplus1]  # r_{i+1}
            # Queue may not be empty: insert new entries at the beginning
            self.queue = [
                dict(config, max_epochs=riplus1)
                for config in best_performing_configurations
            ] + self.queue
            self.observed_error_at_rungs[ri] = []  # Reset
```

```
for rung_index, rung in scheduler.all_observed_error_at_rungs.items():
    errors = [xi[1] for xi in rung]
    d2l.plt.scatter([rung_index] * len(errors), errors)
d2l.plt.xlim(min_number_of_epochs - 0.5, max_number_of_epochs + 0.5)
d2l.plt.xticks(
    np.arange(min_number_of_epochs, max_number_of_epochs + 1),
    np.arange(min_number_of_epochs, max_number_of_epochs + 1)
)
d2l.plt.ylabel("validation error")
d2l.plt.xlabel("epochs")
```

```
for rung_index, rung in scheduler.all_observed_error_at_rungs.items():
    errors = [xi[1] for xi in rung]
    d2l.plt.scatter([rung_index] * len(errors), errors)
d2l.plt.xlim(min_number_of_epochs - 0.5, max_number_of_epochs + 0.5)
d2l.plt.xticks(
    np.arange(min_number_of_epochs, max_number_of_epochs + 1),
    np.arange(min_number_of_epochs, max_number_of_epochs + 1)
)
d2l.plt.ylabel("validation error")
d2l.plt.xlabel("epochs")
```

```
for rung_index, rung in scheduler.all_observed_error_at_rungs.items():
    errors = [xi[1] for xi in rung]
    d2l.plt.scatter([rung_index] * len(errors), errors)
d2l.plt.xlim(min_number_of_epochs - 0.5, max_number_of_epochs + 0.5)
d2l.plt.xticks(
    np.arange(min_number_of_epochs, max_number_of_epochs + 1),
    np.arange(min_number_of_epochs, max_number_of_epochs + 1)
)
d2l.plt.ylabel("validation error")
d2l.plt.xlabel("epochs")
```

構成は、現在の段で観測された性能に基づいて並べ替えられる。

```
@d2l.add_to_class(SuccessiveHalvingScheduler)  #@save
def get_top_n_configurations(self, rung_level, n):
    rung = self.observed_error_at_rungs[rung_level]
    if not rung:
        return []
    sorted_rung = sorted(rung, key=lambda x: x[1])
    return [x[0] for x in sorted_rung[:n]]
```

逐次半減法がニューラルネットワークの例でどのように機能するか見てみよう。ここでは $r_{\min} = 2$、$\eta = 2$、$r_{max} = 10$ とし、段のレベルは $2, 4, 8, 10$ になる。

```
min_number_of_epochs = 2
max_number_of_epochs = 10
eta = 2
num_gpus=1

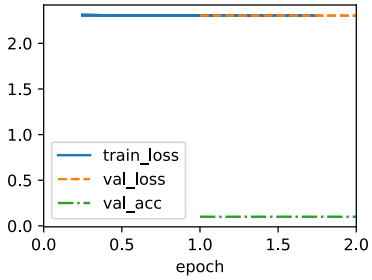
config_space = {
    "learning_rate": stats.loguniform(1e-2, 1),
    "batch_size": stats.randint(32, 256),
}
initial_config = {
    "learning_rate": 0.1,
    "batch_size": 128,
}
```

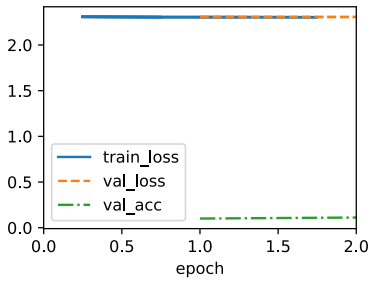
スケジューラを新しい SuccessiveHalvingScheduler に置き換えるだけである。

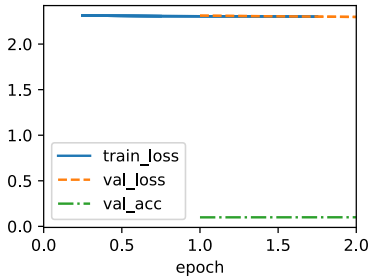
```
searcher = d2l.RandomSearcher(config_space, initial_config=initial_config)
scheduler = SuccessiveHalvingScheduler(
    searcher=searcher,
    eta=eta,
    r_min=min_number_of_epochs,
    r_max=max_number_of_epochs,
)
tuner = d2l.HPOTuner(
    scheduler=scheduler,
    objective=d2l.hpo_objective_lenet,
)
tuner.run(number_of_trials=30)
```

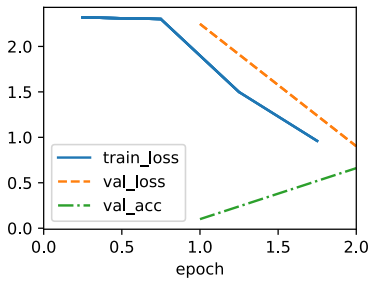
```
    error = 0.1657370924949646, runtime = 67.32617664337158
```

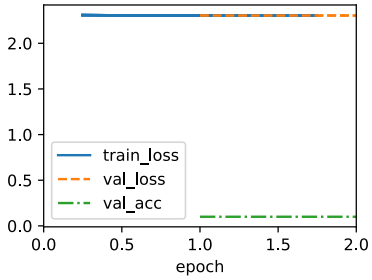
評価したすべての構成の学習曲線を可視化できる。ほとんどの構成は早期に打ち切られ、より性能の良い構成だけが r_{max} まで生き残る。これを、すべての構成に r_{max} を割り当てる通常のランダムサーチと比較せよ。

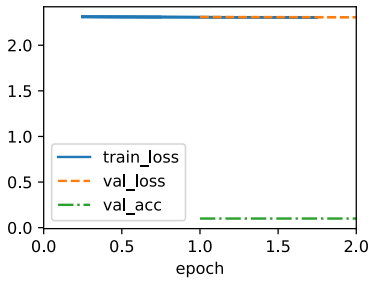

2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
epoch
train_loss
val_loss
val_acc

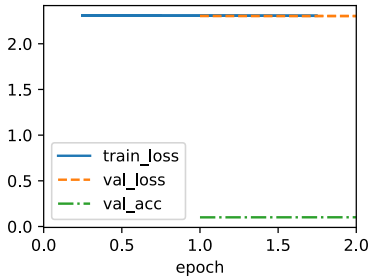

2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
epoch
train_loss
val_loss
val_acc

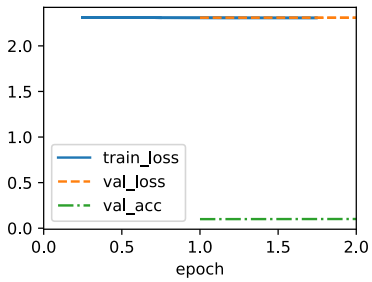

2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
epoch
train_loss
val_loss
val_acc

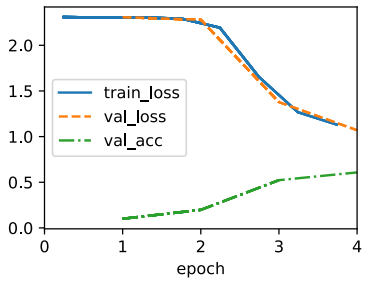

2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
epoch
train_loss
val_loss
val_acc

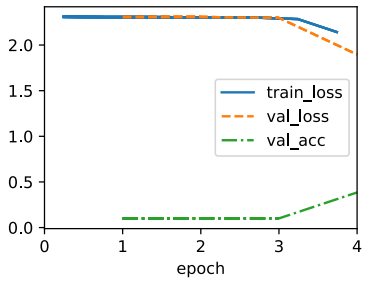

2.0
1.5
1.0
0.5
0.0
train_loss
val_loss
val_acc
0.0
0.5
1.0
1.5
2.0
epoch

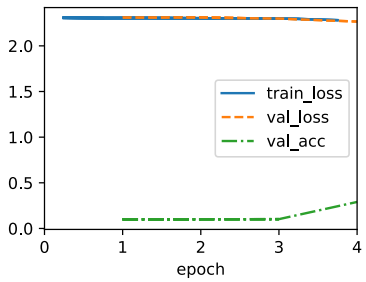

2.0
1.5
1.0
0.5
0.0
train_loss
val_loss
val_acc
0.0
0.5
1.0
1.5
2.0
epoch

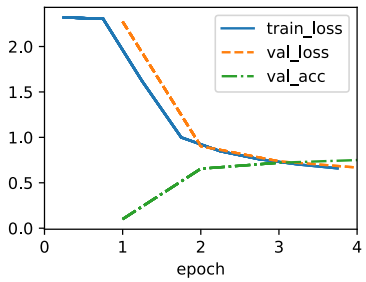

2.0
1.5
1.0
0.5
0.0
train_loss
val_loss
val_acc
0.0
0.5
1.0
1.5
2.0
epoch

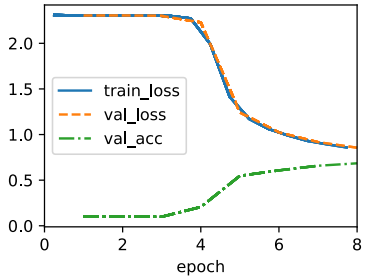

2.0
1.5
1.0
0.5
0.0
train_loss
val_loss
val_acc
0.0
0.5
1.0
1.5
2.0
epoch

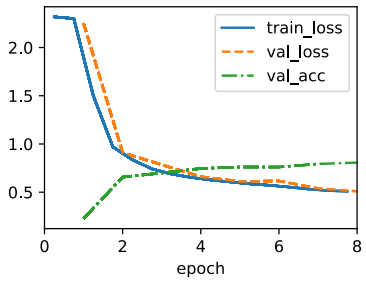

train_loss
val_loss
val_acc
epoch

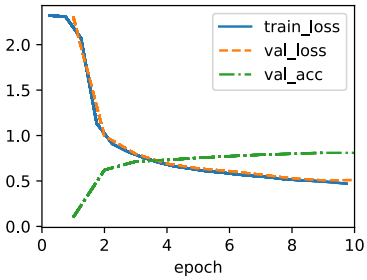

train_loss
val_loss
val_acc
epoch

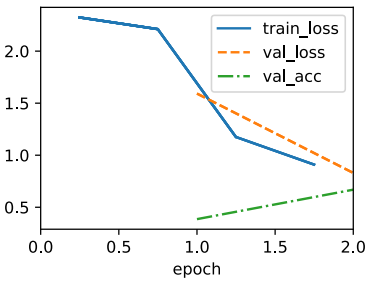

train_loss
val_loss
val_acc
epoch

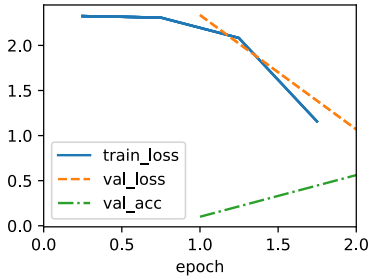

train_loss
val_loss
val_acc
epoch

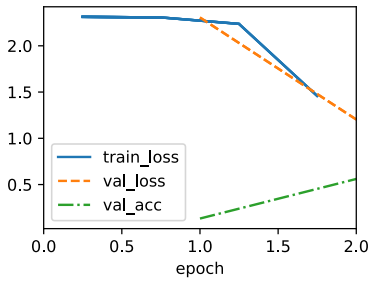

train_loss
val_loss
val_acc
epoch

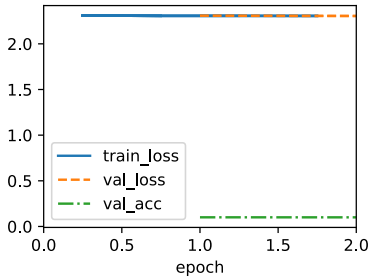

train_loss
val_loss
val_acc
epoch

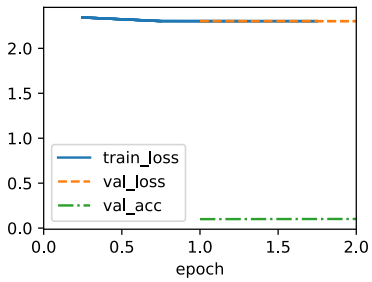

train_loss
val_loss
val_acc
epoch

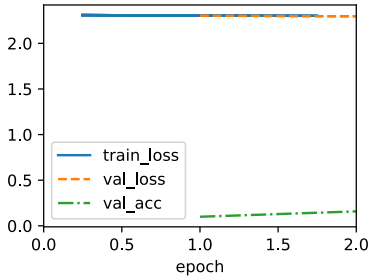

train_loss
val_loss
val_acc
epoch

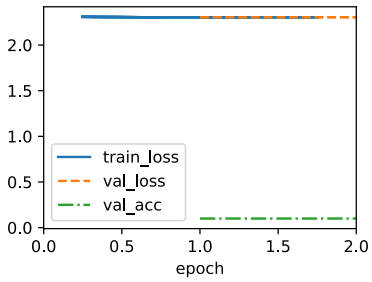

train_loss
val_loss
val_acc
epoch

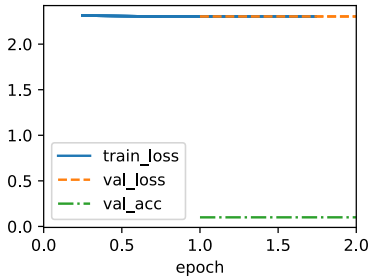

train_loss
val_loss
val_acc
epoch

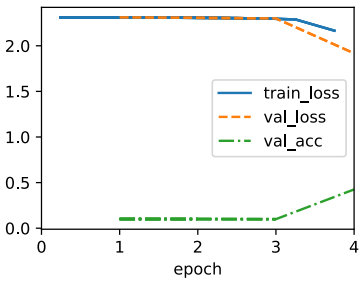

train_loss
val_loss
val_acc
epoch

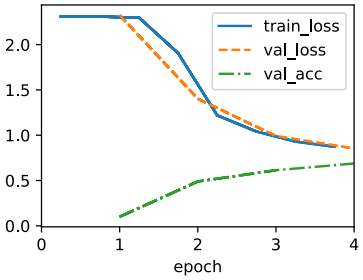

train_loss
val_loss
val_acc
epoch

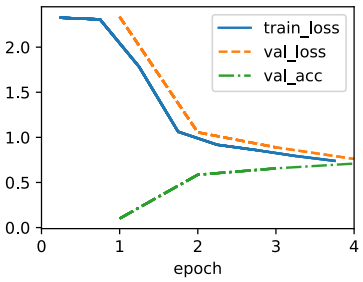

train_loss
val_loss
val_acc
epoch

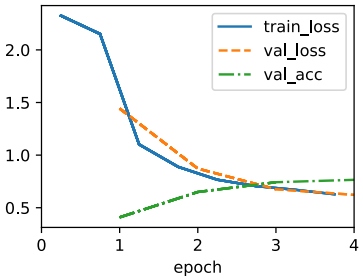

train_loss
val_loss
val_acc
epoch

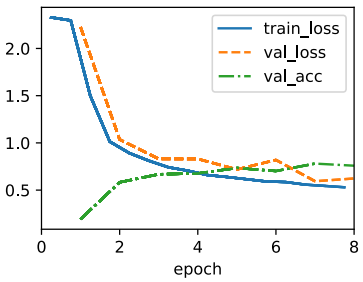

train_loss
val_loss
val_acc
epoch

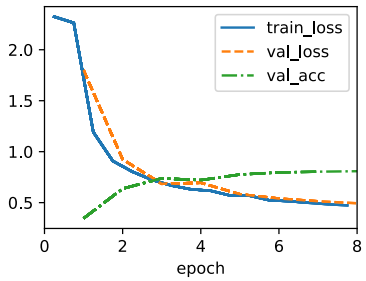

train_loss
val_loss
val_acc
epoch

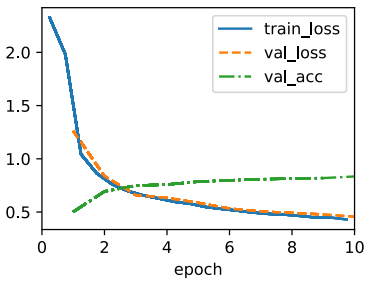

train_loss
val_loss
val_acc
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4
epoch


train_loss
val_loss
val_acc
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4
epoch


train_loss
val_loss
val_acc
2.0
1.5
1.0
0.5
0
1
2
3
4
epoch

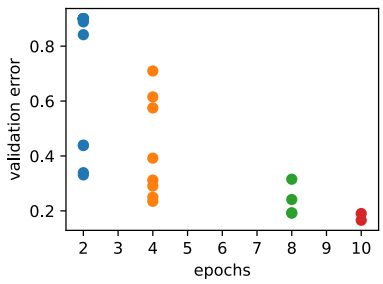

train_loss
val_loss
val_acc
2.0
1.5
1.0
0.5
0
2
4
6
8
epoch






```
for rung_index, rung in scheduler.all_observed_error_at_rungs.items():
    errors = [xi[1] for xi in rung]
    d2l.plt.scatter([rung_index] * len(errors), errors)
d2l.plt.xlim(min_number_of_epochs - 0.5, max_number_of_epochs + 0.5)
d2l.plt.xticks(
    np.arange(min_number_of_epochs, max_number_of_epochs + 1),
    np.arange(min_number_of_epochs, max_number_of_epochs + 1)
)
d2l.plt.ylabel("validation error")
d2l.plt.xlabel("epochs")
```

```
Text(0.5, 0, 'epochs')
```



最後に、SuccessiveHalvingScheduler の実装には少し複雑な点があることに注意せよ。あるワーカーがジョブを実行可能で、現在の段がほぼ埋まっている一方で、別のワーカーがまだ評価中だとする。このワーカーからのメトリック値

がまだないため、次の段を開くための上位 $1/\eta$ の割合を決定できない。一方で、空いているワーカーにジョブを割り当てて、アイドル状態のままにしたくはない。そこで、私たちは新しい逐次半減法のラウンドを開始し、そのワーカーをそこでの最初の試行に割り当てる。ただし、`update` である段が完了したら、新しい構成をキューの先頭に挿入し、次のラウンドの構成よりも優先されるようにしている。

22.4.2 まとめ

この節では、マルチフィデリティ・ハイパーパラメータ最適化の概念を導入した。ここでは、完全なエポック数での検証誤差の代用として、あるエポック数まで学習した後の検証誤差のような、安価に評価できる目的関数の近似にアクセスできると仮定する。マルチフィデリティ・ハイパーパラメータ最適化は、壁時計時間を短縮するだけでなく、HPO に必要な総計算量そのものを削減できる。

単純でありながら効率的なマルチフィデリティ HPO アルゴリズムである逐次半減法を実装し、評価した。

22.5 非同期逐次半減法

Section 22.3 で述べたように、HPO はハイパーパラメータ構成の評価を複数のインスタンスに分散したり、1 つのインスタンス上の複数の CPU/GPU に分散したりすることで高速化が可能である。しかし、ランダムサーチと比較すると、分散環境で逐次半減法（SH）を非同期に実行することは容易ではない。次にどの構成を実行するかを決定する前に、まず現在の rung レベルにあるすべての観測結果を集める必要があるためである。これは各 rung レベルでワーカーを同期させることを意味する。たとえば、最下位の rung レベル $r_{\min}$ では、まず $N = \eta^K$ 個すべての構成を評価し、その後でその $\frac{1}{\eta}$ を次の rung レベルへ昇格させる必要がある。

どの分散システムにおいても、同期は通常ワーカーのアイドル時間を意味する。まず、ハイパーパラメータ構成間で学習時間に大きなばらつきが頻繁に観察される。たとえば、層ごとのフィルタ数がハイパーパラメータである場合、フィルタ数の少ないネットワークは多いネットワークよりも学習が早く終了するため、遅い処理に引きずられてワーカーが待機を余儀なくされる。さらに、ある rung レベルのスロット数がワーカー数の倍数であるとは限らず、その場合には一部のワーカーが 1 バッチ分まるごと遊休となる場合もある。

図 Fig. 22.5.1 は、2 つのワーカーを用いた $\eta = 2$ の同期 SH のスケジューリングを示す。まず Trial-0 と Trial-1 を 1 エポックずつ評価し、それらが終了するとすぐに次の 2 つの trial へ進む。Trial-2 が終了するのを待たなければならず、これには他の trial よりかなり長い時間がかかる。その後で初めて、最良の 2 つの

trial、すなわち Trial-0 と Trial-3 を次の rung レベルへ昇格可能となる。これにより Worker-1 はアイドル状態となる。次に Rung 1 へ進む。ここでも Trial-3 は Trial-0 より時間がかかるため、Worker-0 に追加の待機時間が発生する。Rung-2 に到達すると、最良の trial である Trial-0 だけが残り、1 つのワーカーのみが使用される。その間に Worker-1 が遊休状態を避けるため、SH の多くの実装ではすでに次のラウンドへ進み、最初の rung で新しい trial（たとえば Trial-4）の評価を開始する。

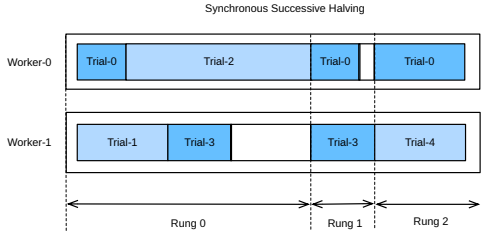


Fig. 22.5.1 2 つのワーカーを用いた同期逐次半減法。

非同期逐次半減法（ASHA）(Li *et al.*, 2018) は、SH を非同期並列シナリオに適応させたものである。ASHA の基本的な考え方は、現在の rung レベルで少なくとも η 個の観測結果が集まったら、すぐに構成を次の rung レベルへ昇格させることである。この決定規則は、最適とは言えない昇格を招く可能性がある。つまり、同じ rung レベルの他の多くと比較すると、後から見ればあまり良くない構成を次の rung レベルへ昇格させてしまう可能性がある。一方で、この方法によりすべての同期点を除去できる。実際には、このような最初の段階での最適でない昇格が性能に与える影響は小さいことが多い。これは、ハイパーパラメータ構成の順位付けが rung レベル間でかなり一貫していることが多いだけでなく、rung が時間とともに大きくなり、そのレベルでのメトリック値の分布をよりよく反映するためである。ワーカーが空いているのに昇格できる構成がない場合は、$r = r_{\min}$、すなわち最初の rung レベルで新しい構成を開始する。

Fig. 22.5.2 は、同じ構成群に対する ASHA のスケジューリングを示す。Trial-1 が終了すると、2 つの trial（すなわち Trial-0 と Trial-1）の結果を集め、そのうち良い方（Trial-0）をすぐに次の rung レベルへ昇格させる。Trial-0 が rung 1 で終了した後、さらに昇格を支えるにはその rung にある trial が不足している。したがって、rung 0 に戻って Trial-3 の評価を継続する。Trial-3 が終了した時点では、Trial-2 はまだ保留中である。この時点で rung 0 では 3 つの trial が評価済みであり、rung 1 では 1 つの trial がすでに評価済みである。Trial-3 は rung 0 で Trial-0 より成績が悪く、かつ $\eta = 2$ なので、まだ新しい trial を昇格させることはできず、代わりに Worker-1 は Trial-4 を最初から開始する。しかし、Trial-2 が終了して Trial-3 より悪いスコアとなった場合、後者が rung 1 へ昇格する。その後、rung 1 で 2 件の評価が集まったので、Trial-0 を rung 2 へ昇格が可能となる。同時に、Worker-1 は rung 0 で新しい trial（すなわち Trial-5）の評価を継続する。

```
from d2l import torch as d2l
import logging
```

(continues on next page)

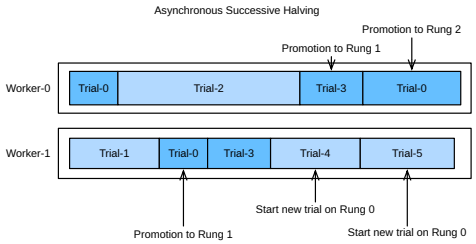


Fig. 22.5.2 2 つのワーカーを用いた非同期逐次半減法（ASHA）。

(continued from previous page)

```
logging.basicConfig(level=logging.INFO)
import matplotlib.pyplot as plt
from syne_tune.config_space import loguniform, randint
from syne_tune.backend.python_backend import PythonBackend
from syne_tune.optimizer.baselines import ASHA
from syne_tune import Tuner, StoppingCriterion
from syne_tune.experiments import load_experiment
```

```
INFO:root:SageMakerBackend is not imported since dependencies are missing. You␣
↪can install them with
   pip install 'syne-tune[extra]'
AWS dependencies are not imported since dependencies are missing. You can␣
↪install them with
   pip install 'syne-tune[aws]'
or (for everything)
   pip install 'syne-tune[extra]'
AWS dependencies are not imported since dependencies are missing. You can␣
↪install them with
   pip install 'syne-tune[aws]'
or (for everything)
   pip install 'syne-tune[extra]'
INFO:root:Ray Tune schedulers and searchers are not imported since␣
↪dependencies are missing. You can install them with
   pip install 'syne-tune[raytune]'
or (for everything)
   pip install 'syne-tune[extra]'
```

22.5.1 目的関数

Section 22.3 と同じ目的関数を *Syne Tune* で利用する。

```
def hpo_objective_lenet_synetune(learning_rate, batch_size, max_epochs):
    from d2l import torch as d2l
    from syne_tune import Reporter

    model = d2l.LeNet(lr=learning_rate, num_classes=10)
    trainer = d2l.HPOTrainer(max_epochs=1, num_gpus=1)
    data = d2l.FashionMNIST(batch_size=batch_size)
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
```

(continues on next page)

(continued from previous page)

```
    report = Reporter()
    for epoch in range(1, max_epochs + 1):
        if epoch == 1:
            # Initialize the state of Trainer
            trainer.fit(model=model, data=data)
        else:
            trainer.fit_epoch()
        validation_error = d2l.numpy(trainer.validation_error().cpu())
        report(epoch=epoch, validation_error=float(validation_error))
```

また、以前と同じ構成空間を用いる。

```
min_number_of_epochs = 2
max_number_of_epochs = 10
eta = 2

config_space = {
    "learning_rate": loguniform(1e-2, 1),
    "batch_size": randint(32, 256),
    "max_epochs": max_number_of_epochs,
}
initial_config = {
    "learning_rate": 0.1,
    "batch_size": 128,
}
```

22.5.2 非同期スケジューラ

まず、trial を並列に評価するワーカー数を定義する。また、総 wall-clock 時間の上限を定めて、ランダムサーチをどれだけ実行するかを指定する必要がある。

```
n_workers = 2  # Needs to be <= the number of available GPUs
max_wallclock_time = 12 * 60  # 12 minutes
```

ASHA を実行するコードは、非同期ランダムサーチで行ったことの単純な変形である。

```
mode = "min"
metric = "validation_error"
resource_attr = "epoch"

scheduler = ASHA(
    config_space,
    metric=metric,
    mode=mode,
    points_to_evaluate=[initial_config],
    max_resource_attr="max_epochs",
    resource_attr=resource_attr,
    grace_period=min_number_of_epochs,
```

(continues on next page)

(continued from previous page)

```
    reduction_factor=eta,
)
```

```
INFO:syne_tune.optimizer.schedulers.fifo:max_resource_level = 10, as inferred␣
↪from config_space
INFO:syne_tune.optimizer.schedulers.fifo:Master random_seed = 4141536384
```

ここで metric と resource_attr は report コールバックで用いられるキー名を指定し、max_resource_attr は目的関数への入力のうちどれが $r_{\max}$ に対応するかを示す。さらに、grace_period は $r_{\min}$ を与え、reduction_factor は η である。以前と同様に Syne Tune を実行可能である（約 12 分を要する）。

```
trial_backend = PythonBackend(
    tune_function=hpo_objective_lenet_synetune,
    config_space=config_space,
)

stop_criterion = StoppingCriterion(max_wallclock_time=max_wallclock_time)
tuner = Tuner(
    trial_backend=trial_backend,
    scheduler=scheduler,
    stop_criterion=stop_criterion,
    n_workers=n_workers,
    print_update_interval=int(max_wallclock_time * 0.6),
)
tuner.run()
```

```
INFO:syne_tune.tuner:results of trials will be saved on /home/ci/syne-tune/
↪python-entrypoint-2023-08-18-07-20-36-065
INFO:root:Detected 4 GPUs
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.1 --batch_size 128 --max_epochs 10 --tune_function_root␣
↪/home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/tune_function --
↪tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_checkpoint_dir /
↪home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/0/checkpoints
INFO:syne_tune.tuner:(trial 0) - scheduled config {'learning_rate': 0.1,
↪'batch_size': 128, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.016393465295512565 --batch_size 109 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪1/checkpoints
INFO:syne_tune.tuner:(trial 1) - scheduled config {'learning_rate': 0.
↪016393465295512565, 'batch_size': 109, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.8536433102570197 --batch_size 249 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
```

(continues on next page)

(continued from previous page)

```
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪2/checkpoints
INFO:syne_tune.tuner:(trial 2) - scheduled config {'learning_rate': 0.
↪8536433102570197, 'batch_size': 249, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.02238278285575226 --batch_size 241 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪3/checkpoints
INFO:syne_tune.tuner:(trial 3) - scheduled config {'learning_rate': 0.
↪02238278285575226, 'batch_size': 241, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 2 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.02265916956093905 --batch_size 104 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪4/checkpoints
INFO:syne_tune.tuner:(trial 4) - scheduled config {'learning_rate': 0.
↪02265916956093905, 'batch_size': 104, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.08152606473767814 --batch_size 144 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪5/checkpoints
INFO:syne_tune.tuner:(trial 5) - scheduled config {'learning_rate': 0.
↪08152606473767814, 'batch_size': 144, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.03286717277851087 --batch_size 230 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪6/checkpoints
INFO:syne_tune.tuner:(trial 6) - scheduled config {'learning_rate': 0.
↪03286717277851087, 'batch_size': 230, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.2767625103368729 --batch_size 210 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪7/checkpoints
INFO:syne_tune.tuner:(trial 7) - scheduled config {'learning_rate': 0.
↪2767625103368729, 'batch_size': 210, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.20515181717902709 --batch_size 136 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
```

(continues on next page)

(continued from previous page)

```
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪8/checkpoints
INFO:syne_tune.tuner:(trial 8) - scheduled config {'learning_rate': 0.
↪20515181717902709, 'batch_size': 136, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.01205522192914604 --batch_size 61 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪9/checkpoints
INFO:syne_tune.tuner:(trial 9) - scheduled config {'learning_rate': 0.
↪01205522192914604, 'batch_size': 61, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.06812982725901803 --batch_size 225 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪10/checkpoints
INFO:syne_tune.tuner:(trial 10) - scheduled config {'learning_rate': 0.
↪06812982725901803, 'batch_size': 225, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.15408580663479204 --batch_size 184 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪11/checkpoints
INFO:syne_tune.tuner:(trial 11) - scheduled config {'learning_rate': 0.
↪15408580663479204, 'batch_size': 184, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.03319086816357744 --batch_size 215 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪12/checkpoints
INFO:syne_tune.tuner:(trial 12) - scheduled config {'learning_rate': 0.
↪03319086816357744, 'batch_size': 215, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.5871483249986997 --batch_size 131 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪13/checkpoints
INFO:syne_tune.tuner:(trial 13) - scheduled config {'learning_rate': 0.
↪5871483249986997, 'batch_size': 131, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.04405320051007572 --batch_size 64 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
```

(continues on next page)

(continued from previous page)

```
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪14/checkpoints
INFO:syne_tune.tuner:(trial 14) - scheduled config {'learning_rate': 0.
↪04405320051007572, 'batch_size': 64, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.15163221599975787 --batch_size 51 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪15/checkpoints
INFO:syne_tune.tuner:(trial 15) - scheduled config {'learning_rate': 0.
↪15163221599975787, 'batch_size': 51, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.02010844681019881 --batch_size 197 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪16/checkpoints
INFO:syne_tune.tuner:(trial 16) - scheduled config {'learning_rate': 0.
↪02010844681019881, 'batch_size': 197, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.030927553495777638 --batch_size 131 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪17/checkpoints
INFO:syne_tune.tuner:(trial 17) - scheduled config {'learning_rate': 0.
↪030927553495777638, 'batch_size': 131, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.06962532946104748 --batch_size 228 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪18/checkpoints
INFO:syne_tune.tuner:(trial 18) - scheduled config {'learning_rate': 0.
↪06962532946104748, 'batch_size': 228, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.21744469415997278 --batch_size 240 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪19/checkpoints
INFO:syne_tune.tuner:(trial 19) - scheduled config {'learning_rate': 0.
↪21744469415997278, 'batch_size': 240, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.06936041631165406 --batch_size 106 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
```

(continues on next page)

(continued from previous page)

```
↪20/checkpoints
INFO:syne_tune.tuner:(trial 20) - scheduled config {'learning_rate': 0.
↪06936041631165406, 'batch_size': 106, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.42010437996754035 --batch_size 181 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪21/checkpoints
INFO:syne_tune.tuner:(trial 21) - scheduled config {'learning_rate': 0.
↪42010437996754035, 'batch_size': 181, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.618264683792025 --batch_size 230 --max_epochs 10 --tune_
↪function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪22/checkpoints
INFO:syne_tune.tuner:(trial 22) - scheduled config {'learning_rate': 0.
↪618264683792025, 'batch_size': 230, 'max_epochs': 10}
INFO:syne_tune.tuner:tuning status (last metric is reported)
 trial_id     status  iter  learning_rate  batch_size  max_epochs  epoch ␣
↪validation_error  worker-time
        0    Stopped    10       0.100000         128          10     10       ␣
↪   0.268790    71.102878
        1    Stopped     2       0.016393         109          10      2       ␣
↪   0.899935    13.896730
        2  Completed    10       0.853643         249          10     10       ␣
↪   0.182956    66.696980
        3    Stopped     2       0.022383         241          10      2       ␣
↪   0.877944    12.613355
        4    Stopped     2       0.022659         104          10      2       ␣
↪   0.900327    14.157292
        5    Stopped     2       0.081526         144          10      2       ␣
↪   0.900298    13.433055
        6    Stopped     4       0.032867         230          10      4       ␣
↪   0.915217    24.955152
        7    Stopped    10       0.276763         210          10     10       ␣
↪   0.233890    60.715368
        8    Stopped     8       0.205152         136          10      8       ␣
↪   0.325495    47.716144
        9    Stopped     2       0.012055          61          10      2       ␣
↪   0.899991    16.423611
       10    Stopped     4       0.068130         225          10      4       ␣
↪   0.899753    23.875174
       11    Stopped     4       0.154086         184          10      4       ␣
↪   0.453026    23.736529
       12    Stopped     2       0.033191         215          10      2       ␣
↪   0.899906    12.479618
       13    Stopped     8       0.587148         131          10      8       ␣
↪   0.375490    51.454139
       14    Stopped     2       0.044053          64          10      2       ␣
↪   0.899881    15.800886
       15    Stopped    10       0.151632          51          10     10       ␣
```

(continues on next page)

(continued from previous page)

```
↪   0.188712   106.156809
      16    Stopped     2       0.020108         197          10      2       ␣
↪   0.900156    11.984264
      17    Stopped     2       0.030928         131          10      2       ␣
↪   0.900078    11.954435
      18    Stopped     2       0.069625         228          10      2       ␣
↪   0.900124    11.464556
      19    Stopped     2       0.217445         240          10      2       ␣
↪   0.899950    11.154259
      20    Stopped     2       0.069360         106          10      2       ␣
↪   0.899923    12.920582
      21 InProgress     8       0.420104         181          10      8       ␣
↪   0.234927    51.076726
      22 InProgress     7       0.618265         230          10      7       ␣
↪   0.240343    43.615625
2 trials running, 21 finished (1 until the end), 436.97s wallclock-time

INFO:syne_tune.tuner:Trial trial_id 21 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.16400286598134425 --batch_size 87 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪23/checkpoints
INFO:syne_tune.tuner:(trial 23) - scheduled config {'learning_rate': 0.
↪16400286598134425, 'batch_size': 87, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 22 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.16603894676453146 --batch_size 132 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪24/checkpoints
INFO:syne_tune.tuner:(trial 24) - scheduled config {'learning_rate': 0.
↪16603894676453146, 'batch_size': 132, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.12085342681840087 --batch_size 161 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪25/checkpoints
INFO:syne_tune.tuner:(trial 25) - scheduled config {'learning_rate': 0.
↪12085342681840087, 'batch_size': 161, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.020613727673297316 --batch_size 69 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪26/checkpoints
INFO:syne_tune.tuner:(trial 26) - scheduled config {'learning_rate': 0.
↪020613727673297316, 'batch_size': 69, 'max_epochs': 10}
```

(continues on next page)

(continued from previous page)

```
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.7417151784363999 --batch_size 57 --max_epochs 10 --tune_
↪function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪27/checkpoints
INFO:syne_tune.tuner:(trial 27) - scheduled config {'learning_rate': 0.
↪7417151784363999, 'batch_size': 57, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 24 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.5851545371142973 --batch_size 43 --max_epochs 10 --tune_
↪function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪28/checkpoints
INFO:syne_tune.tuner:(trial 28) - scheduled config {'learning_rate': 0.
↪5851545371142973, 'batch_size': 43, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 27 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.012313981089293918 --batch_size 167 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪29/checkpoints
INFO:syne_tune.tuner:(trial 29) - scheduled config {'learning_rate': 0.
↪012313981089293918, 'batch_size': 167, 'max_epochs': 10}
INFO:syne_tune.tuner:Trial trial_id 28 completed.
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.18350672552566955 --batch_size 49 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪30/checkpoints
INFO:syne_tune.tuner:(trial 30) - scheduled config {'learning_rate': 0.
↪18350672552566955, 'batch_size': 49, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.2726593425130573 --batch_size 128 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪31/checkpoints
INFO:syne_tune.tuner:(trial 31) - scheduled config {'learning_rate': 0.
↪2726593425130573, 'batch_size': 128, 'max_epochs': 10}
INFO:root:running subprocess with command: /usr/bin/python /home/ci/.local/lib/
↪python3.8/site-packages/syne_tune/backend/python_backend/python_entrypoint.
↪py --learning_rate 0.6661438697130982 --batch_size 106 --max_epochs 10 --
↪tune_function_root /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-
↪065/tune_function --tune_function_hash 638f9e6b337c8f0d289f336d612fb672 --st_
↪checkpoint_dir /home/ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065/
↪32/checkpoints
```

(continues on next page)

(continued from previous page)

```
INFO:syne_tune.tuner:(trial 32) - scheduled config {'learning_rate': 0.
↪6661438697130982, 'batch_size': 106, 'max_epochs': 10}
INFO:syne_tune.stopping_criterion:reaching max wallclock time (720), stopping␣
↪there.
INFO:syne_tune.tuner:Stopping trials that may still be running.
INFO:syne_tune.tuner:Tuning finished, results of trials can be found on /home/
↪ci/syne-tune/python-entrypoint-2023-08-18-07-20-36-065
--------------------
Resource summary (last result is reported):
 trial_id     status  iter  learning_rate  batch_size  max_epochs  epoch ␣
↪validation_error  worker-time
        0    Stopped    10       0.100000         128          10   10.0      ␣
↪   0.268790    71.102878
        1    Stopped     2       0.016393         109          10    2.0      ␣
↪   0.899935    13.896730
        2  Completed    10       0.853643         249          10   10.0      ␣
↪   0.182956    66.696980
        3    Stopped     2       0.022383         241          10    2.0      ␣
↪   0.877944    12.613355
        4    Stopped     2       0.022659         104          10    2.0      ␣
↪   0.900327    14.157292
        5    Stopped     2       0.081526         144          10    2.0      ␣
↪   0.900298    13.433055
        6    Stopped     4       0.032867         230          10    4.0      ␣
↪   0.915217    24.955152
        7    Stopped    10       0.276763         210          10   10.0      ␣
↪   0.233890    60.715368
        8    Stopped     8       0.205152         136          10    8.0      ␣
↪   0.325495    47.716144
        9    Stopped     2       0.012055          61          10    2.0      ␣
↪   0.899991    16.423611
       10    Stopped     4       0.068130         225          10    4.0      ␣
↪   0.899753    23.875174
       11    Stopped     4       0.154086         184          10    4.0      ␣
↪   0.453026    23.736529
       12    Stopped     2       0.033191         215          10    2.0      ␣
↪   0.899906    12.479618
       13    Stopped     8       0.587148         131          10    8.0      ␣
↪   0.375490    51.454139
       14    Stopped     2       0.044053          64          10    2.0      ␣
↪   0.899881    15.800886
       15    Stopped    10       0.151632          51          10   10.0      ␣
↪   0.188712   106.156809
       16    Stopped     2       0.020108         197          10    2.0      ␣
↪   0.900156    11.984264
       17    Stopped     2       0.030928         131          10    2.0      ␣
↪   0.900078    11.954435
       18    Stopped     2       0.069625         228          10    2.0      ␣
↪   0.900124    11.464556
       19    Stopped     2       0.217445         240          10    2.0      ␣
↪   0.899950    11.154259
       20    Stopped     2       0.069360         106          10    2.0      ␣
↪   0.899923    12.920582
       21  Completed    10       0.420104         181          10   10.0      ␣
↪   0.213122    64.977173
```

(continues on next page)

(continued from previous page)

```
      22  Completed    10       0.618265         230          10   10.0       ␣
↪   0.190514    63.500903
      23    Stopped     4       0.164003          87          10    4.0       ␣
↪   0.345055    29.901440
      24  Completed    10       0.166039         132          10   10.0       ␣
↪   0.230829    72.114988
      25    Stopped     2       0.120853         161          10    2.0       ␣
↪   0.900627    13.288419
      26    Stopped     2       0.020614          69          10    2.0       ␣
↪   0.899995    16.607311
      27  Completed    10       0.741715          57          10   10.0       ␣
↪   0.124957    88.111582
      28  Completed    10       0.585155          43          10   10.0       ␣
↪   0.132764   103.217489
      29    Stopped     2       0.012314         167          10    2.0       ␣
↪   0.899969    12.800696
      30 InProgress     7       0.183507          49          10    7.0       ␣
↪   0.176705    66.790888
      31    Stopped    10       0.272659         128          10   10.0       ␣
↪   0.201048    71.773432
      32 InProgress     0       0.666144         106          10      -       ␣
↪         -         -
2 trials running, 31 finished (6 until the end), 723.55s wallclock-time

validation_error: best 0.12495654821395874 for trial-id 27
---------------------
```

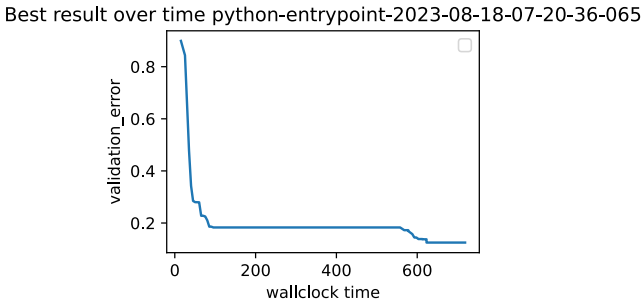
ここでは、性能の低い trial を早期停止する ASHA の変種を実行していることに留意されたい。これは、Section 22.4.1 における実装とは異なる。そちらでは、各学習ジョブは固定の max_epochs で開始される。後者の場合、10 エポックすべてに到達する性能の良い trial は、まず 1 エポック、次に 2 エポック、その次に 4 エポック、さらに 8 エポックと、毎回最初から学習し直す必要がある。この種の一時停止・再開型のスケジューリングは、各エポック後に学習状態をチェックポイント化することで効率的に実装可能であるが、ここではその追加の複雑さは避ける。実験が終了したら、結果を取得してプロット可能である。

```
d2l.set_figsize()
e = load_experiment(tuner.name)
e.plot()
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note␣
↪that artists whose label start with an underscore are ignored when legend()␣
↪is called with no argument.
```

22.5.3 最適化過程の可視化

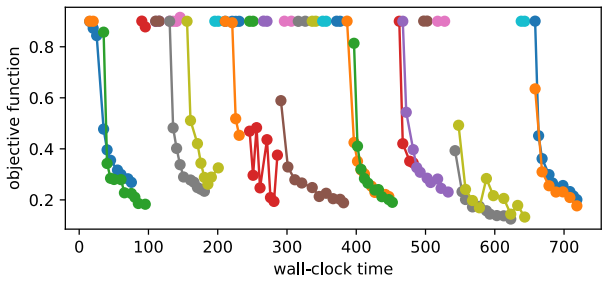
再度、各 trial の学習曲線を可視化する（プロット中の各色は 1 つの trial を表す）。これを Section 22.3 の非同期ランダムサーチと比較されたい。Section 22.4 で逐次半減法について述べたように、trial の大半は 1 エポックまたは 2 エポック



（$r_{\min}$ または $\eta * r_{\min}$）で停止する。しかし、各 trial は 1 エポックあたりに必要な時間が異なるため、同じ時点では停止しない。もし ASHA ではなく標準的な逐次半減法を実行する場合、構成を次の rung レベルへ昇格させる前に、ワーカーを同期させる必要がある。

```
d2l.set_figsize([6, 2.5])
results = e.results
for trial_id in results.trial_id.unique():
    df = results[results["trial_id"] == trial_id]
    d2l.plt.plot(
        df["st_tuner_time"],
        df["validation_error"],
        marker="o"
    )
d2l.plt.xlabel("wall-clock time")
d2l.plt.ylabel("objective function")
```

```
Text(0, 0.5, 'objective function')
```



22.5.4 まとめ

ランダムサーチと比べると、逐次半減法を非同期分散環境で実行するのはそれほど自明ではない。同期点を避けるために、多少誤ったものを昇格させることになっても、できるだけ速く構成を次の rung レベルへ昇格させる。実際には、これは通常それほど大きな悪影響を及ぼさない。非同期スケジューリングと同

期スケジューリングの差による利得は、最適でない意思決定による損失よりも通常ははるかに大きいからである。

23 生成的敵対的ネットワーク

23.1 生成的敵対的ネットワーク

本書の大部分では、予測をどのように行うかについて述べてきた。何らかの形で、深層ニューラルネットワークを用いてデータ例からラベルへの写像を学習してきた。この種の学習は識別学習と呼ばれる。たとえば、猫の写真と犬の写真を見分けられるようにしたい、という場合である。分類器と回帰器はどちらも識別学習の例である。そして、バックプロパゲーションで訓練されたニューラルネットワークは、大規模で複雑なデータセットにおける識別学習について、私たちが知っていたことを一変させた。高解像度画像での分類精度は、わずか 5〜6 年で役に立たない水準から人間並みの水準へと（いくつかの注意点はあるものの）向上した。深層ニューラルネットワークが驚くほど高い性能を発揮する、他のあらゆる識別タスクについて、ここで改めて長々と述べることは控えよう。

しかし、機械学習は識別タスクを解くだけではない。たとえば、大規模なデータセットがあってもラベルがない場合、そのデータの特徴を簡潔に捉えるモデルを学習したいことがある。そのようなモデルがあれば、訓練データの分布に似た合成データ例をサンプリングできる。たとえば、顔写真の大規模なコーパスがあるとき、同じデータセットから来たとしても不自然ではないような、新しい写実的な画像を生成したいかもしれない。この種の学習は生成モデリングと呼ばれる。

最近まで、斬新な写実的画像を合成する方法は存在しなかった。しかし、識別学習における深層ニューラルネットワークの成功が、新たな可能性を切り開いた。ここ 3 年ほどの大きな潮流の 1 つは、通常は教師あり学習問題とは考えない問題における課題を克服するために、識別的な深層ネットワークを応用することであった。再帰型ニューラルネットワークによる言語モデルは、識別ネットワーク（次の文字を予測するように訓練されたもの）を用い、訓練後には生成モデルとして機能しうる例である。

2014 年、画期的な論文が生成的敵対的ネットワーク（GAN）を導入した (Goodfellow *et al.*, 2014)。これは、識別モデルの力を活用して優れた生成モデルを得るための、巧妙で新しい方法である。GAN の核心にあるのは、偽データと本物のデータを見分けられないなら、そのデータ生成器は優れている、という考え方

である。統計学では、これは 2 標本検定と呼ばれる。すなわち、データセット $X = \{x_1, \ldots, x_n\}$ と $X' = \{x'_1, \ldots, x'_n\}$ が同じ分布から抽出されたかどうかを答えるための検定である。多くの統計学の論文と GAN の主な違いは、後者がこの考え方を構成的に用いる点にある。言い換えると、単に「この 2 つのデータセットは同じ分布から来たようには見えない」と言うモデルを訓練するのではなく、2 標本検定[131] を生成モデルへの学習信号として利用するのである。これにより、実データに似たものを生成するまでデータ生成器を改善できる。少なくとも、最先端の深層ニューラルネットワークであっても、それを欺ける必要がある。

131



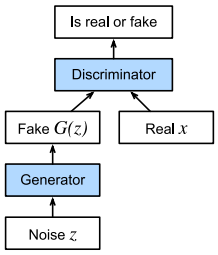


Fig. 23.1.1 生成的敵対的ネットワーク

GAN のアーキテクチャを Fig. 23.1.1 に示す。見てのとおり、GAN のアーキテクチャには 2 つの要素がある。まず、実物そっくりのデータを生成できる可能性のある装置（たとえば深層ネットワークであるが、実際にはゲームのレンダリングエンジンのような何でもよい）を必要とする。画像を扱うなら画像を生成する必要がある。音声を扱うなら音声系列を生成する必要があり、以下同様である。これを生成器ネットワークと呼ぶ。第 2 の要素は識別器ネットワークである。これは偽データと本物のデータを互いに見分けようとする。両者は互いに競争関係にある。生成器ネットワークは識別器ネットワークを欺こうとする。その時点で、識別器ネットワークは新しい偽データに適応する。この情報は、今度は生成器ネットワークを改善するために使われ、これが繰り返される。

識別器は、入力 x が本物（実データ由来）か偽物（生成器由来）かを判別する 2 値分類器である。通常、識別器は入力 $\mathbf{x}$ に対してスカラー予測 $o \in \mathbb{R}$ を出力する。たとえば、隠れサイズ 1 の全結合層を用い、その後にシグモイド関数を適用して予測確率 $D(\mathbf{x}) = 1/(1 + e^{-o})$ を得る。真のデータのラベルを 1、偽データのラベルを 0 とする。識別器は交差エントロピー損失を最小化するように訓練する。すなわち、

$$\min_D \{-y \log D(\mathbf{x}) - (1 - y) \log(1 - D(\mathbf{x}))\}, \tag{23.1.1}$$

生成器については、まずランダム性の源からパラメータ $\mathbf{z} \in \mathbb{R}^d$ をサンプルする。たとえば、正規分布 $\mathbf{z} \sim \mathcal{N}(0, 1)$ である。$\mathbf{z}$ は潜在変数と呼ばれることがよくある。そして関数を適用して $\mathbf{x}' = G(\mathbf{z})$ を生成する。生成器の目標は、識別器を欺いて $\mathbf{x}' = G(\mathbf{z})$ を真のデータとして分類させること、すなわち $D(G(\mathbf{z})) \approx 1$ とすることである。言い換えると、与えられた識別器 D に対して、$y = 0$ のときの交差エントロピー損失を最大化するように生成器 G のパラメータを更新す

る。すなわち、

$$\max_G \{-(1-y)\log(1-D(G(\mathbf{z})))\} = \max_G \{-\log(1-D(G(\mathbf{z})))\}. \tag{23.1.2}$$

生成器が完全にうまくいけば、$D(\mathbf{x}') \approx 1$ となるので、上の損失は 0 に近くなり、その結果、識別器が十分に進歩するための勾配が小さすぎるという問題が生じる。そこで通常は、次の損失を最小化する。

$$\min_G \{-y\log(D(G(\mathbf{z})))\} = \min_G \{-\log(D(G(\mathbf{z})))\}, \tag{23.1.3}$$

これは、$\mathbf{x}' = G(\mathbf{z})$ を識別器に入力する一方で、ラベル $y = 1$ を与えるのと同じである。

要するに、D と G は次の包括的な目的関数を用いた「ミニマックス」ゲームを行っている。

$$\min_D \max_G \{-E_{x\sim\text{Data}} \log D(\mathbf{x}) - E_{z\sim\text{Noise}} \log(1-D(G(\mathbf{z})))\}. \tag{23.1.4}$$

GAN の応用の多くは画像の文脈にある。デモンストレーションの目的として、まずはもっと単純な分布の当てはめに満足することにしよう。GAN を使ってガウス分布のパラメータを推定する、世界で最も非効率な推定器を構築するとどうなるかを示す。始めよう。

```
%matplotlib inline
from d2l import torch as d2l
import torch
from torch import nn
```

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

```
batch_size = 8
data_iter = d2l.load_array((data,), batch_size)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

23.1.1 いくつかの「実データ」を生成する

これは世界で最もつまらない例になるので、単にガウス分布から抽出したデータを生成する。

```
X = d2l.normal(0.0, 1, (1000, 2))
A = d2l.tensor([[1, 2], [-0.1, 0.5]])
b = d2l.tensor([1, 2])
data = d2l.matmul(X, A) + b
```

```
X = d2l.normal(0.0, 1, (1000, 2))
A = d2l.tensor([[1, 2], [-0.1, 0.5]])
b = d2l.tensor([1, 2])
data = d2l.matmul(X, A) + b
```

```
[07:04:39] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

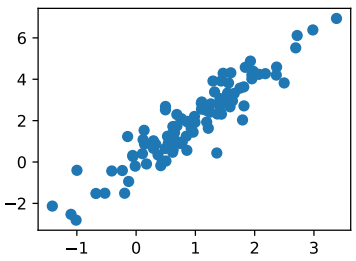
```
lr_D, lr_G, latent_dim, num_epochs = 0.05, 0.005, 2, 20
train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G,
      latent_dim, d2l.numpy(data[:100]))
```

```
X = d2l.normal((1000, 2), 0.0, 1)
A = d2l.tensor([[1, 2], [-0.1, 0.5]])
b = d2l.tensor([1, 2], tf.float32)
data = d2l.matmul(X, A) + b
```

得られたものを見てみよう。これは、平均 b、共分散行列 $A^T A$ をもつガウス分布が、かなり任意な形で平行移動されたものになるはずである。

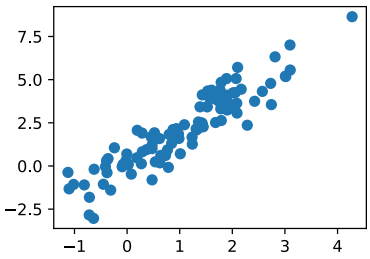
```
d2l.set_figsize()
d2l.plt.scatter(d2l.numpy(data[:100, 0]), d2l.numpy(data[:100, 1]));
print(f'The covariance matrix is\n{d2l.matmul(A.T, A)}')
```

```
The covariance matrix is
tensor([[1.0100, 1.9500],
        [1.9500, 4.2500]])
```



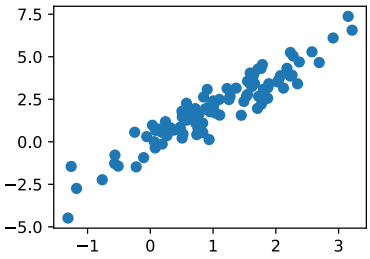

```
d2l.set_figsize()
d2l.plt.scatter(d2l.numpy(data[:100, 0]), d2l.numpy(data[:100, 1]));
print(f'The covariance matrix is\n{d2l.matmul(A.T, A)}')
```

```
The covariance matrix is
[[1.01 1.95]
 [1.95 4.25]]
```




```
d2l.set_figsize()
d2l.plt.scatter(d2l.numpy(data[:100, 0]), d2l.numpy(data[:100, 1]));
print(f'The covariance matrix is\n{tf.matmul(A, A, transpose_a=True)}')
```

```
The covariance matrix is
[[1.01 1.95]
 [1.95 4.25]]
```




```
batch_size = 8
data_iter = d2l.load_array((data,), batch_size)
```

23.1.2 生成器

生成器ネットワークは、可能な限り最も単純なネットワーク、つまり 1 層の線形モデルにする。これは、線形ネットワークをガウス型のデータ生成器で駆動するからである。したがって、完全に偽装するためのパラメータを学習するだけでよいのである。

```
net_G = nn.Sequential(nn.Linear(2, 2))
```

```
net_G = nn.Sequential()
net_G.add(nn.Dense(2))
```

```
net_G = tf.keras.layers.Dense(2)
```

23.1.3 識別器

識別器については、もう少し識別的にいきよう。少し面白くするために、3 層の MLP を使う。

```
net_D = nn.Sequential(
    nn.Linear(2, 5), nn.Tanh(),
    nn.Linear(5, 3), nn.Tanh(),
    nn.Linear(3, 1))
```

```
net_D = nn.Sequential()
net_D.add(nn.Dense(5, activation='tanh'),
          nn.Dense(3, activation='tanh'),
          nn.Dense(1))
```

```
net_D = tf.keras.models.Sequential([
    tf.keras.layers.Dense(5, activation="tanh", input_shape=(2,)),
    tf.keras.layers.Dense(3, activation="tanh"),
    tf.keras.layers.Dense(1)
])
```

23.1.4 訓練

まず、識別器を更新する関数を定義する。

```
#@save
def update_D(X, Z, net_D, net_G, loss, trainer_D):
    """ 識別器を更新する。"""
    batch_size = X.shape[0]
    ones = torch.ones((batch_size,), device=X.device)
    zeros = torch.zeros((batch_size,), device=X.device)
    trainer_D.zero_grad()
    real_Y = net_D(X)
    fake_X = net_G(Z)
    # `net_G` の勾配を計算する必要はないので、勾配計算から切り離す。
    fake_Y = net_D(fake_X.detach())
    loss_D = (loss(real_Y, ones.reshape(real_Y.shape)) +
              loss(fake_Y, zeros.reshape(fake_Y.shape))) / 2
    loss_D.backward()
    trainer_D.step()
    return loss_D
```

```
#@save
def update_D(X, Z, net_D, net_G, loss, trainer_D):
    """ 識別器を更新する。"""
    batch_size = X.shape[0]
```

(continues on next page)

(continued from previous page)

```
    ones = np.ones((batch_size,), ctx=X.ctx)
    zeros = np.zeros((batch_size,), ctx=X.ctx)
    with autograd.record():
        real_Y = net_D(X)
        fake_X = net_G(Z)
        # `net_G` の勾配を計算する必要はないので、勾配計算から切り離す。
        fake_Y = net_D(fake_X.detach())
        loss_D = (loss(real_Y, ones) + loss(fake_Y, zeros)) / 2
    loss_D.backward()
    trainer_D.step(batch_size)
    return float(loss_D.sum())
```

```
#@save
def update_D(X, Z, net_D, net_G, loss, optimizer_D):
    """ 識別器を更新する。"""
    batch_size = X.shape[0]
    ones = tf.ones((batch_size,)) # 実データに対応するラベル
    zeros = tf.zeros((batch_size,)) # 偽データに対応するラベル
    # `net_G` の勾配を計算する必要はないので、GradientTape の外に置く
    fake_X = net_G(Z)
    with tf.GradientTape() as tape:
        real_Y = net_D(X)
        fake_Y = net_D(fake_X)
        # PyTorch の BCEWithLogitsLoss に合わせるため、損失に batch_size を掛け
る
        loss_D = (loss(ones, tf.squeeze(real_Y)) + loss(
            zeros, tf.squeeze(fake_Y))) * batch_size / 2
    grads_D = tape.gradient(loss_D, net_D.trainable_variables)
    optimizer_D.apply_gradients(zip(grads_D, net_D.trainable_variables))
    return loss_D
```

生成器も同様に更新する。ここでは交差エントロピー損失を再利用するが、偽データのラベルを 0 から 1 に変更する。

```
#@save
def update_G(Z, net_D, net_G, loss, trainer_G):
    """ 生成器を更新する。"""
    batch_size = Z.shape[0]
    ones = torch.ones((batch_size,), device=Z.device)
    trainer_G.zero_grad()
    # 計算を節約するために `update_D` での `fake_X` を再利用してもよい
    fake_X = net_G(Z)
    # `net_D` が変化するので `fake_Y` の再計算が必要
    fake_Y = net_D(fake_X)
    loss_G = loss(fake_Y, ones.reshape(fake_Y.shape))
    loss_G.backward()
    trainer_G.step()
    return loss_G
```

```
#@save
def update_G(Z, net_D, net_G, loss, trainer_G):
```

(continues on next page)

(continued from previous page)

```
    """ 生成器を更新する。"""
    batch_size = Z.shape[0]
    ones = np.ones((batch_size,), ctx=Z.ctx)
    with autograd.record():
        # 計算を節約するために `update_D` での `fake_X` を再利用してもよい
        fake_X = net_G(Z)
        # `net_D` が変化するので `fake_Y` の再計算が必要
        fake_Y = net_D(fake_X)
        loss_G = loss(fake_Y, ones)
    loss_G.backward()
    trainer_G.step(batch_size)
    return float(loss_G.sum())
```

```
#@save
def update_G(Z, net_D, net_G, loss, optimizer_G):
    """ 生成器を更新する。"""
    batch_size = Z.shape[0]
    ones = tf.ones((batch_size,))
    with tf.GradientTape() as tape:
        # 計算を節約するために `update_D` での `fake_X` を再利用してもよい
        fake_X = net_G(Z)
        # `net_D` が変化するので `fake_Y` の再計算が必要
        fake_Y = net_D(fake_X)
        # PyTorch の BCEWithLogits loss に合わせるため、損失に batch_size を掛
ける
        loss_G = loss(ones, tf.squeeze(fake_Y)) * batch_size
    grads_G = tape.gradient(loss_G, net_G.trainable_variables)
    optimizer_G.apply_gradients(zip(grads_G, net_G.trainable_variables))
    return loss_G
```

識別器と生成器の両方は、交差エントロピー損失を用いた 2 値ロジスティック回帰を行う。訓練過程を安定させるために Adam を使う。各反復では、まず識別器を更新し、その後に生成器を更新する。両方の損失と生成例を可視化する。

```
def train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data):
    loss = nn.BCEWithLogitsLoss(reduction='sum')
    for w in net_D.parameters():
        nn.init.normal_(w, 0, 0.02)
    for w in net_G.parameters():
        nn.init.normal_(w, 0, 0.02)
    trainer_D = torch.optim.Adam(net_D.parameters(), lr=lr_D)
    trainer_G = torch.optim.Adam(net_G.parameters(), lr=lr_G)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(num_epochs):
        # 1 エポック訓練する
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
```

(continues on next page)

(continued from previous page)

```
        for (X,) in data_iter:
            batch_size = X.shape[0]
            Z = torch.normal(0, 1, size=(batch_size, latent_dim))
            metric.add(update_D(X, Z, net_D, net_G, loss, trainer_D),
                       update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # 生成例を可視化する
        Z = torch.normal(0, 1, size=(100, latent_dim))
        fake_X = net_G(Z).detach().numpy()
        animator.axes[1].cla()
        animator.axes[1].scatter(data[:, 0], data[:, 1])
        animator.axes[1].scatter(fake_X[:, 0], fake_X[:, 1])
        animator.axes[1].legend(['real', 'generated'])
        # 損失を表示する
        loss_D, loss_G = metric[0]/metric[2], metric[1]/metric[2]
        animator.add(epoch + 1, (loss_D, loss_G))
    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec')
```

```
def train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data):
    loss = gluon.loss.SigmoidBCELoss()
    net_D.initialize(init=init.Normal(0.02), force_reinit=True)
    net_G.initialize(init=init.Normal(0.02), force_reinit=True)
    trainer_D = gluon.Trainer(net_D.collect_params(),
                              'adam', {'learning_rate': lr_D})
    trainer_G = gluon.Trainer(net_G.collect_params(),
                              'adam', {'learning_rate': lr_G})
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(num_epochs):
        # 1 エポック訓練する
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for X in data_iter:
            batch_size = X.shape[0]
            Z = np.random.normal(0, 1, size=(batch_size, latent_dim))
            metric.add(update_D(X, Z, net_D, net_G, loss, trainer_D),
                       update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # 生成例を可視化する
        Z = np.random.normal(0, 1, size=(100, latent_dim))
        fake_X = net_G(Z).asnumpy()
        animator.axes[1].cla()
        animator.axes[1].scatter(data[:, 0], data[:, 1])
        animator.axes[1].scatter(fake_X[:, 0], fake_X[:, 1])
        animator.axes[1].legend(['real', 'generated'])
        # 損失を表示する
        loss_D, loss_G = metric[0]/metric[2], metric[1]/metric[2]
        animator.add(epoch + 1, (loss_D, loss_G))
    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec')
```

```
def train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data):
    loss = tf.keras.losses.BinaryCrossentropy(
        from_logits=True, reduction=tf.keras.losses.Reduction.SUM)
    for w in net_D.trainable_variables:
        w.assign(tf.random.normal(mean=0, stddev=0.02, shape=w.shape))
    for w in net_G.trainable_variables:
        w.assign(tf.random.normal(mean=0, stddev=0.02, shape=w.shape))
    optimizer_D = tf.keras.optimizers.Adam(learning_rate=lr_D)
    optimizer_G = tf.keras.optimizers.Adam(learning_rate=lr_G)
    animator = d2l.Animator(
        xlabel="epoch", ylabel="loss", xlim=[1, num_epochs], nrows=2,
        figsize=(5, 5), legend=["discriminator", "generator"])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(num_epochs):
        # 1 エポック訓練する
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for (X,) in data_iter:
            batch_size = X.shape[0]
            Z = tf.random.normal(
                mean=0, stddev=1, shape=(batch_size, latent_dim))
            metric.add(update_D(X, Z, net_D, net_G, loss, optimizer_D),
                       update_G(Z, net_D, net_G, loss, optimizer_G),
                       batch_size)
        # 生成例を可視化する
        Z = tf.random.normal(mean=0, stddev=1, shape=(100, latent_dim))
        fake_X = net_G(Z)
        animator.axes[1].cla()
        animator.axes[1].scatter(data[:, 0], data[:, 1])
        animator.axes[1].scatter(fake_X[:, 0], fake_X[:, 1])
        animator.axes[1].legend(["real", "generated"])

        # 損失を表示する
        loss_D, loss_G = metric[0] / metric[2], metric[1] / metric[2]
        animator.add(epoch + 1, (loss_D, loss_G))

    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec')
```

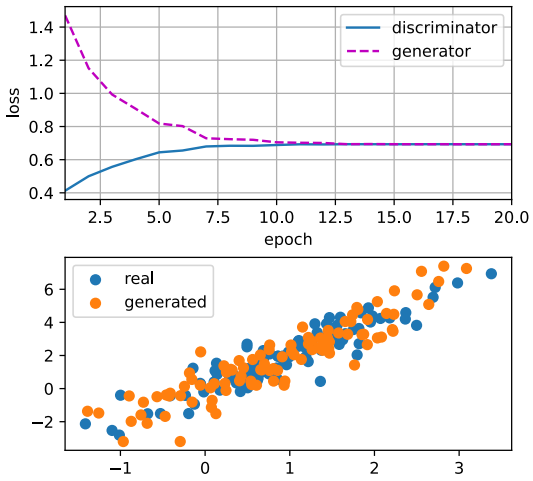
ここで、ガウス分布を当てはめるためのハイパーパラメータを指定する。

```
lr_D, lr_G, latent_dim, num_epochs = 0.05, 0.005, 2, 20
train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G,
      latent_dim, d2l.numpy(data[:100]))
```

```
loss_D 0.693, loss_G 0.693, 1167.8 examples/sec
```

23.1.5 まとめ

- 生成的敵対的ネットワーク（GAN）は、生成器と識別器という 2 つの深層ネットワークから構成される。
- 生成器は、識別器を欺くために、実画像にできるだけ近い画像を生成するよ



うに、交差エントロピー損失を最大化する。すなわち、$\max \log(D(\mathbf{x}'))$ である。

- 識別器は、生成画像と真の画像を区別しようとして、交差エントロピー損失を最小化する。すなわち、$\min -y \log D(\mathbf{x}) - (1-y) \log(1 - D(\mathbf{x}))$ である。

23.1.6 演習

- 生成器が勝つ、すなわち有限サンプル上で識別器が 2 つの分布を区別できなくなるような平衡は存在するだろうか？

23.2 Deep Convolutional Generative Adversarial Networks

Section 23.1 では、GAN がどのように動作するかの基本的な考え方を紹介した。GAN は、一様分布や正規分布のような、単純でサンプリングしやすい分布からサンプルを取り出し、それをデータセットの分布に一致しているように見えるサンプルへと変換できることを示した。2 次元ガウス分布を一致させる例でも要点は伝わるが、特に刺激的というわけではない。

この節では、GAN を用いて写実的な画像を生成する方法を示す。ここでは、Radford *et al.* (2015) で導入された deep convolutional GAN（DCGAN）を基にモデルを構築する。識別的なコンピュータビジョン問題で大きな成功を収めてきた畳み込みアーキテクチャを取り入れ、GAN を通じてそれらが写実的な画像生成に活用できることを示す。

```
from d2l import torch as d2l
import torch
import torchvision
from torch import nn
import warnings
```

```
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

```
from d2l import tensorflow as d2l
import tensorflow as tf
```

23.2.1 ポケモンデータセット

ここで使用するデータセットは、pokemondb[132] から取得したポケモンのスプライト画像のコレクションである。まず、このデータセットをダウンロード、展開、読み込みする。

132

```
#@save
d2l.DATA_HUB['pokemon'] = (d2l.DATA_URL + 'pokemon.zip',
                           'c065c0e2593b8b161a2d7873e42418bf6a21106c')

data_dir = d2l.download_extract('pokemon')
pokemon = torchvision.datasets.ImageFolder(data_dir)
```

```
Downloading ../data/pokemon.zip from http://d2l-data.s3-accelerate.amazonaws.
↪com/pokemon.zip...
```

```
#@save
d2l.DATA_HUB['pokemon'] = (d2l.DATA_URL + 'pokemon.zip',
                           'c065c0e2593b8b161a2d7873e42418bf6a21106c')

data_dir = d2l.download_extract('pokemon')
pokemon = gluon.data.vision.datasets.ImageFolderDataset(data_dir)
```

```
Downloading ../data/pokemon.zip from http://d2l-data.s3-accelerate.amazonaws.
↪com/pokemon.zip...
```

```
#@save
d2l.DATA_HUB['pokemon'] = (d2l.DATA_URL + 'pokemon.zip',
                           'c065c0e2593b8b161a2d7873e42418bf6a21106c')

data_dir = d2l.download_extract('pokemon')
```

(continues on next page)

(continued from previous page)

```
batch_size = 256
pokemon = tf.keras.preprocessing.image_dataset_from_directory(
    data_dir, batch_size=batch_size, image_size=(64, 64))
```

```
Downloading ../data/pokemon.zip from http://d2l-data.s3-accelerate.amazonaws.
↪com/pokemon.zip...
Found 40597 files belonging to 721 classes.
```

各画像を 64×64 にリサイズする。ToTensor 変換は画素値を $[0, 1]$ に写像し、生成器は tanh 関数を用いて $[-1, 1]$ の出力を得る。したがって、値の範囲を合わせるために、平均 0.5、標準偏差 0.5 でデータを正規化する。

```
batch_size = 256
transformer = torchvision.transforms.Compose([
    torchvision.transforms.Resize((64, 64)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(0.5, 0.5)
])
pokemon.transform = transformer
data_iter = torch.utils.data.DataLoader(
    pokemon, batch_size=batch_size,
    shuffle=True, num_workers=d2l.get_dataloader_workers())
```

```
batch_size = 256
transformer = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.Resize(64),
    gluon.data.vision.transforms.ToTensor(),
    gluon.data.vision.transforms.Normalize(0.5, 0.5)
])
data_iter = gluon.data.DataLoader(
    pokemon.transform_first(transformer), batch_size=batch_size,
    shuffle=True, num_workers=d2l.get_dataloader_workers())
```

```
def transform_func(X):
    X = X / 255.
    X = (X - 0.5) / (0.5)
    return X

# For TF>=2.4 use `num_parallel_calls = tf.data.AUTOTUNE`
data_iter = pokemon.map(lambda x, y: (transform_func(x), y),
                        num_parallel_calls=tf.data.experimental.AUTOTUNE)
data_iter = data_iter.cache().shuffle(buffer_size=1000).prefetch(
    buffer_size=tf.data.experimental.AUTOTUNE)
```

最初の 20 枚の画像を可視化してみよう。

```
warnings.filterwarnings('ignore')
d2l.set_figsize((4, 4))
for X, y in data_iter:
```

(continues on next page)

(continued from previous page)

```
    imgs = X[:20,:,:,:].permute(0, 2, 3, 1)/2+0.5
    d2l.show_images(imgs, num_rows=4, num_cols=5)
    break
```

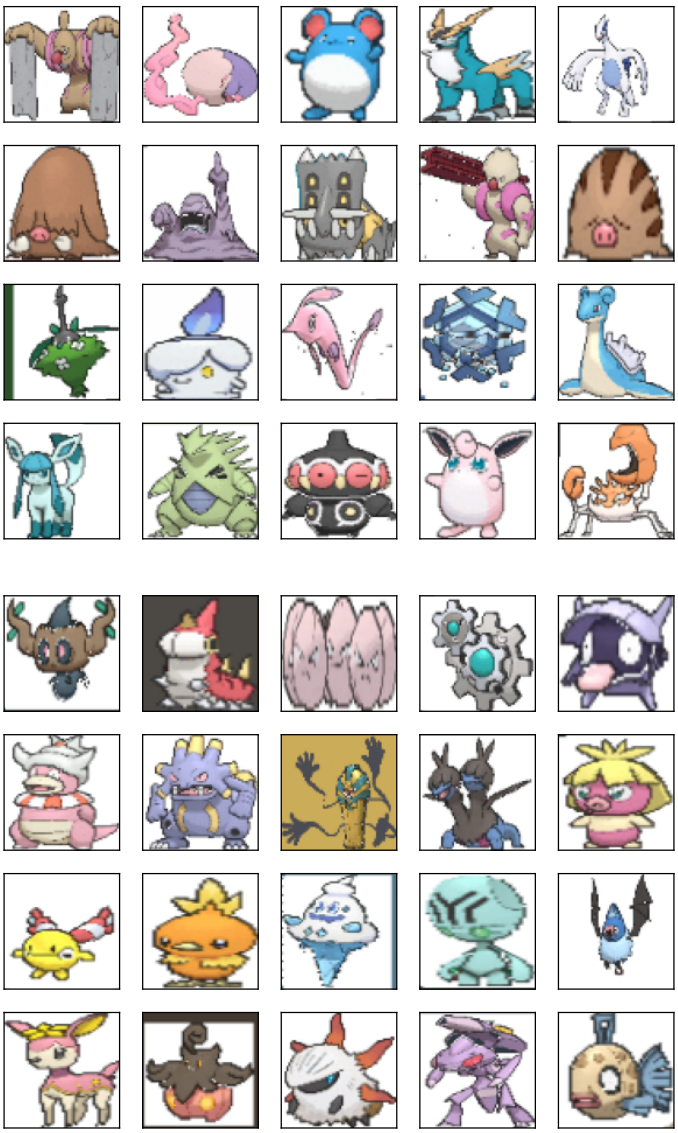


```
d2l.set_figsize((4, 4))
for X, y in data_iter:
    imgs = X[:20,:,:,:].transpose(0, 2, 3, 1)/2+0.5
    d2l.show_images(imgs, num_rows=4, num_cols=5)
    break
```

```
[08:01:50] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
[08:01:50] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
[08:01:50] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
[08:01:50] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
[08:01:50] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
d2l.set_figsize(figsize=(4, 4))
for X, y in data_iter.take(1):
    imgs = X[:20, :, :, :] / 2 + 0.5
    d2l.show_images(imgs, num_rows=4, num_cols=5)
```

23.2.2 生成器



生成器は、ノイズ変数 $\mathbf{z} \in \mathbb{R}^d$（長さ d のベクトル）を、幅と高さが 64×64 の RGB 画像へ写像する必要がある。Section 17.11 では、転置畳み込み層（Section 17.10 を参照）を用いて入力サイズを拡大する全畳み込みネットワークを紹介した。生成器の基本ブロックは、転置畳み込み層の後にバッチ正規化と ReLU 活性化を続けたものである。

```
class G_block(nn.Module):
    def __init__(self, out_channels, in_channels=3, kernel_size=4, strides=2,
                 padding=1, **kwargs):
        super(G_block, self).__init__(**kwargs)
```

(continues on next page)

(continued from previous page)

```
        self.conv2d_trans = nn.ConvTranspose2d(in_channels, out_channels,
                                    kernel_size, strides, padding, bias=False)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.activation = nn.ReLU()

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d_trans(X)))
```

```
class G_block(nn.Block):
    def __init__(self, channels, kernel_size=4,
                 strides=2, padding=1, **kwargs):
        super(G_block, self).__init__(**kwargs)
        self.conv2d_trans = nn.Conv2DTranspose(
            channels, kernel_size, strides, padding, use_bias=False)
        self.batch_norm = nn.BatchNorm()
        self.activation = nn.Activation('relu')

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d_trans(X)))
```

```
class G_block(tf.keras.layers.Layer):
    def __init__(self, out_channels, kernel_size=4, strides=2, padding="same",
                 **kwargs):
        super().__init__(**kwargs)
        self.conv2d_trans = tf.keras.layers.Conv2DTranspose(
            out_channels, kernel_size, strides, padding, use_bias=False)
        self.batch_norm = tf.keras.layers.BatchNormalization()
        self.activation = tf.keras.layers.ReLU()

    def call(self, X):
        return self.activation(self.batch_norm(self.conv2d_trans(X)))
```

デフォルトでは、転置畳み込み層は $k_h = k_w = 4$ のカーネル、$s_h = s_w = 2$ のストライド、$p_h = p_w = 1$ のパディングを用いる。入力形状が $n_h' \times n_w' = 16 \times 16$ のとき、生成器ブロックは入力の幅と高さを 2 倍にする。

$$\begin{aligned} n_h' \times n_w' &= [(n_h k_h - (n_h - 1)(k_h - s_h) - 2p_h] \times [(n_w k_w - (n_w - 1)(k_w - s_w) - 2p_w] \\ &= [(k_h + s_h(n_h - 1) - 2p_h] \times [(k_w + s_w(n_w - 1) - 2p_w] \\ &= [(4 + 2 \times (16 - 1) - 2 \times 1] \times [(4 + 2 \times (16 - 1) - 2 \times 1] \\ &= 32 \times 32. \end{aligned} \tag{23.2.1}$$

```
x = torch.zeros((2, 3, 16, 16))
g_blk = G_block(20)
g_blk(x).shape
```

```
torch.Size([2, 20, 32, 32])
```

```
x = np.zeros((2, 3, 16, 16))
g_blk = G_block(20)
g_blk.initialize()
g_blk(x).shape
```

```
(2, 20, 32, 32)
```

```
x = tf.zeros((2, 16, 16, 3))  # Channel last convention
g_blk = G_block(20)
g_blk(x).shape
```

```
TensorShape([2, 32, 32, 20])
```

転置畳み込み層を 4×4 カーネル、1×1 ストライド、ゼロパディングに変更するとする。入力サイズが 1×1 のとき、出力の幅と高さはそれぞれ 3 ずつ増加する。

```
x = torch.zeros((2, 3, 1, 1))
g_blk = G_block(20, strides=1, padding=0)
g_blk(x).shape
```

```
torch.Size([2, 20, 4, 4])
```

```
x = np.zeros((2, 3, 1, 1))
g_blk = G_block(20, strides=1, padding=0)
g_blk.initialize()
g_blk(x).shape
```

```
(2, 20, 4, 4)
```

```
x = tf.zeros((2, 1, 1, 3))
# `padding="valid"` corresponds to no padding
g_blk = G_block(20, strides=1, padding="valid")
g_blk(x).shape
```

```
TensorShape([2, 4, 4, 20])
```

生成器は 4 つの基本ブロックからなり、入力の幅と高さを 1 から 32 へ増やする。同時に、まず潜在変数を 64×8 チャネルへ射影し、その後は各段階でチャネル数を半分にする。最後に、出力を生成するために転置畳み込み層を用いる。これにより幅と高さをさらに 2 倍にして、望ましい 64×64 の形状に合わせ、チャネル数を 3 に減らする。tanh 活性化関数を適用して、出力値を $(-1, 1)$ の範囲に射影する。

```
n_G = 64
net_G = nn.Sequential(
    G_block(in_channels=100, out_channels=n_G*8,
            strides=1, padding=0),                  # Output: (64 * 8, 4, 4)
    G_block(in_channels=n_G*8, out_channels=n_G*4), # Output: (64 * 4, 8, 8)
    G_block(in_channels=n_G*4, out_channels=n_G*2), # Output: (64 * 2, 16, 16)
    G_block(in_channels=n_G*2, out_channels=n_G),   # Output: (64, 32, 32)
    nn.ConvTranspose2d(in_channels=n_G, out_channels=3,
                       kernel_size=4, stride=2, padding=1, bias=False),
    nn.Tanh())  # Output: (3, 64, 64)
```

```
n_G = 64
net_G = nn.Sequential()
net_G.add(G_block(n_G*8, strides=1, padding=0),  # Output: (64 * 8, 4, 4)
          G_block(n_G*4),  # Output: (64 * 4, 8, 8)
          G_block(n_G*2),  # Output: (64 * 2, 16, 16)
          G_block(n_G),    # Output: (64, 32, 32)
          nn.Conv2DTranspose(
              3, kernel_size=4, strides=2, padding=1, use_bias=False,
              activation='tanh'))  # Output: (3, 64, 64)
```

```
n_G = 64
net_G = tf.keras.Sequential([
    # Output: (4, 4, 64 * 8)
    G_block(out_channels=n_G*8, strides=1, padding="valid"),
    G_block(out_channels=n_G*4), # Output: (8, 8, 64 * 4)
    G_block(out_channels=n_G*2), # Output: (16, 16, 64 * 2)
    G_block(out_channels=n_G), # Output: (32, 32, 64)
    # Output: (64, 64, 3)
    tf.keras.layers.Conv2DTranspose(
        3, kernel_size=4, strides=2, padding="same", use_bias=False,
        activation="tanh")
])
```

生成器の出力形状を確認するために、100 次元の潜在変数を生成する。

```
x = torch.zeros((1, 100, 1, 1))
net_G(x).shape
```

```
torch.Size([1, 3, 64, 64])
```

```
x = np.zeros((1, 100, 1, 1))
net_G.initialize()
net_G(x).shape
```

```
(1, 3, 64, 64)
```

```
x = tf.zeros((1, 1, 1, 100))
net_G(x).shape
```

```
TensorShape([1, 64, 64, 3])
```

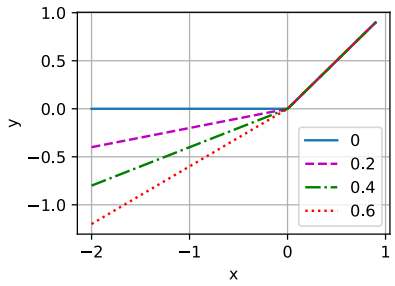
23.2.3 識別器

識別器は通常の畳み込みネットワークであるが、活性化関数として leaky ReLU を用いる点が異なる。$\alpha \in [0, 1]$ に対して、その定義は次のとおりである。

$$\text{leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}. \tag{23.2.2}$$

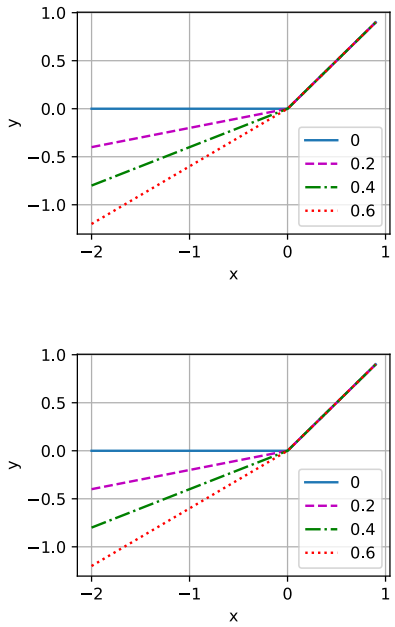
見てのとおり、$\alpha = 0$ なら通常の ReLU、$\alpha = 1$ なら恒等関数である。$\alpha \in (0, 1)$ では、leaky ReLU は負の入力に対しても 0 でない出力を返す非線形関数である。これは、ニューロンが常に負の値を出力してしまい、ReLU の勾配が 0 であるために学習が進まなくなる「dying ReLU」問題を解決することを目的としている。

```
alphas = [0, .2, .4, .6, .8, 1]
x = d2l.arange(-2, 1, 0.1)
Y = [d2l.numpy(nn.LeakyReLU(alpha)(x)) for alpha in alphas]
d2l.plot(d2l.numpy(x), Y, 'x', 'y', alphas)
```




```
alphas = [0, .2, .4, .6, .8, 1]
x = d2l.arange(-2, 1, 0.1)
Y = [d2l.numpy(nn.LeakyReLU(alpha)(x)) for alpha in alphas]
d2l.plot(d2l.numpy(x), Y, 'x', 'y', alphas)
```

```
alphas = [0, .2, .4, .6, .8, 1]
x = tf.range(-2, 1, 0.1)
Y = [tf.keras.layers.LeakyReLU(alpha)(x).numpy() for alpha in alphas]
d2l.plot(x.numpy(), Y, 'x', 'y', alphas)
```



識別器の基本ブロックは、畳み込み層の後にバッチ正規化層と leaky ReLU 活性化を続けたものである。畳み込み層のハイパーパラメータは、生成器ブロックの転置畳み込み層と似ている。

```
class D_block(nn.Module):
    def __init__(self, out_channels, in_channels=3, kernel_size=4, strides=2,
                 padding=1, alpha=0.2, **kwargs):
        super(D_block, self).__init__(**kwargs)
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size,
                                strides, padding, bias=False)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.activation = nn.LeakyReLU(alpha, inplace=True)

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d(X)))
```

```
class D_block(nn.Block):
    def __init__(self, channels, kernel_size=4, strides=2,
                 padding=1, alpha=0.2, **kwargs):
        super(D_block, self).__init__(**kwargs)
        self.conv2d = nn.Conv2D(
            channels, kernel_size, strides, padding, use_bias=False)
        self.batch_norm = nn.BatchNorm()
        self.activation = nn.LeakyReLU(alpha)

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d(X)))
```

```
class D_block(tf.keras.layers.Layer):
    def __init__(self, out_channels, kernel_size=4, strides=2, padding="same",
                 alpha=0.2, **kwargs):
        super().__init__(**kwargs)
        self.conv2d = tf.keras.layers.Conv2D(out_channels, kernel_size,
                                             strides, padding, use_bias=False)
        self.batch_norm = tf.keras.layers.BatchNormalization()
        self.activation = tf.keras.layers.LeakyReLU(alpha)

    def call(self, X):
        return self.activation(self.batch_norm(self.conv2d(X)))
```

デフォルト設定の基本ブロックは、Section 10.3 で示したように、入力の幅と高さを半分にする。たとえば、入力形状が $n_h = n_w = 16$、カーネル形状が $k_h = k_w = 4$、ストライド形状が $s_h = s_w = 2$、パディング形状が $p_h = p_w = 1$ のとき、出力形状は次のようになる。

$$\begin{aligned} n_h' \times n_w' &= \lfloor (n_h - k_h + 2p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + 2p_w + s_w)/s_w \rfloor \\ &= \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \times \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \\ &= 8 \times 8. \end{aligned} \tag{23.2.3}$$

```
x = torch.zeros((2, 3, 16, 16))
d_blk = D_block(20)
d_blk(x).shape
```

```
torch.Size([2, 20, 8, 8])
```

```
x = np.zeros((2, 3, 16, 16))
d_blk = D_block(20)
d_blk.initialize()
d_blk(x).shape
```

```
(2, 20, 8, 8)
```

```
x = tf.zeros((2, 16, 16, 3))
d_blk = D_block(20)
d_blk(x).shape
```

```
TensorShape([2, 8, 8, 20])
```

識別器は生成器の鏡像である。

```
n_D = 64
net_D = nn.Sequential(
    D_block(n_D),  # Output: (64, 32, 32)
```

(continues on next page)

(continued from previous page)

```
    D_block(in_channels=n_D, out_channels=n_D*2),  # Output: (64 * 2, 16, 16)
    D_block(in_channels=n_D*2, out_channels=n_D*4),  # Output: (64 * 4, 8, 8)
    D_block(in_channels=n_D*4, out_channels=n_D*8),  # Output: (64 * 8, 4, 4)
    nn.Conv2d(in_channels=n_D*8, out_channels=1,
              kernel_size=4, bias=False))  # Output: (1, 1, 1)
```

```
n_D = 64
net_D = nn.Sequential()
net_D.add(D_block(n_D),   # Output: (64, 32, 32)
          D_block(n_D*2),  # Output: (64 * 2, 16, 16)
          D_block(n_D*4),  # Output: (64 * 4, 8, 8)
          D_block(n_D*8),  # Output: (64 * 8, 4, 4)
          nn.Conv2D(1, kernel_size=4, use_bias=False))  # Output: (1, 1, 1)
```

```
n_D = 64
net_D = tf.keras.Sequential([
    D_block(n_D), # Output: (32, 32, 64)
    D_block(out_channels=n_D*2), # Output: (16, 16, 64 * 2)
    D_block(out_channels=n_D*4), # Output: (8, 8, 64 * 4)
    D_block(out_channels=n_D*8), # Outupt: (4, 4, 64 * 64)
    # Output: (1, 1, 1)
    tf.keras.layers.Conv2D(1, kernel_size=4, use_bias=False)
])
```

最終層には出力チャネル数 1 の畳み込み層を用いて、単一の予測値を得る。

```
x = torch.zeros((1, 3, 64, 64))
net_D(x).shape
```

```
torch.Size([1, 1, 1, 1])
```

```
x = np.zeros((1, 3, 64, 64))
net_D.initialize()
net_D(x).shape
```

```
(1, 1, 1, 1)
```

```
x = tf.zeros((1, 64, 64, 3))
net_D(x).shape
```

```
TensorShape([1, 1, 1, 1])
```

23.2.4 学習

Section 23.1 の基本的な GAN と比べると、生成器と識別器は互いに似ているため、両者に同じ学習率を用いる。さらに、Adam（Section 15.10）の β_1 を 0.9 から 0.5 に変更する。これは、生成器と識別器が互いに競い合うことで勾配が急速に変化するため、過去の勾配の指数移動平均であるモーメンタムの滑らかさを下げるものである。加えて、ランダムに生成されるノイズ Z は 4 次元テンソルであり、計算の高速化のために GPU を使用する。

```
def train(net_D, net_G, data_iter, num_epochs, lr, latent_dim,
          device=d2l.try_gpu()):
    loss = nn.BCEWithLogitsLoss(reduction='sum')
    for w in net_D.parameters():
        nn.init.normal_(w, 0, 0.02)
    for w in net_G.parameters():
        nn.init.normal_(w, 0, 0.02)
    net_D, net_G = net_D.to(device), net_G.to(device)
    trainer_hp = {'lr': lr, 'betas': [0.5,0.999]}
    trainer_D = torch.optim.Adam(net_D.parameters(), **trainer_hp)
    trainer_G = torch.optim.Adam(net_G.parameters(), **trainer_hp)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(1, num_epochs + 1):
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for X, _ in data_iter:
            batch_size = X.shape[0]
            Z = torch.normal(0, 1, size=(batch_size, latent_dim, 1, 1))
            X, Z = X.to(device), Z.to(device)
            metric.add(d2l.update_D(X, Z, net_D, net_G, loss, trainer_D),
                       d2l.update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # Show generated examples
        Z = torch.normal(0, 1, size=(21, latent_dim, 1, 1), device=device)
        # Normalize the synthetic data to N(0, 1)
        fake_x = net_G(Z).permute(0, 2, 3, 1) / 2 + 0.5
        imgs = torch.cat(
            [torch.cat([
                fake_x[i * 7 + j].cpu().detach() for j in range(7)], dim=1)
             for i in range(len(fake_x)//7)], dim=0)
        animator.axes[1].cla()
        animator.axes[1].imshow(imgs)
        # Show the losses
        loss_D, loss_G = metric[0] / metric[2], metric[1] / metric[2]
        animator.add(epoch, (loss_D, loss_G))
    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec on {str(device)}')
```

```
def train(net_D, net_G, data_iter, num_epochs, lr, latent_dim,
          device=d2l.try_gpu()):
    loss = gluon.loss.SigmoidBCELoss()
    net_D.initialize(init=init.Normal(0.02), force_reinit=True, ctx=device)
```

(continues on next page)

(continued from previous page)

```
    net_G.initialize(init=init.Normal(0.02), force_reinit=True, ctx=device)
    trainer_hp = {'learning_rate': lr, 'beta1': 0.5}
    trainer_D = gluon.Trainer(net_D.collect_params(), 'adam', trainer_hp)
    trainer_G = gluon.Trainer(net_G.collect_params(), 'adam', trainer_hp)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(1, num_epochs + 1):
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for X, _ in data_iter:
            batch_size = X.shape[0]
            Z = np.random.normal(0, 1, size=(batch_size, latent_dim, 1, 1))
            X, Z = X.as_in_ctx(device), Z.as_in_ctx(device),
            metric.add(d2l.update_D(X, Z, net_D, net_G, loss, trainer_D),
                       d2l.update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # Show generated examples
        Z = np.random.normal(0, 1, size=(21, latent_dim, 1, 1), ctx=device)
        # Normalize the synthetic data to N(0, 1)
        fake_x = net_G(Z).transpose(0, 2, 3, 1) / 2 + 0.5
        imgs = np.concatenate(
            [np.concatenate([fake_x[i * 7 + j] for j in range(7)], axis=1)
             for i in range(len(fake_x)//7)], axis=0)
        animator.axes[1].cla()
        animator.axes[1].imshow(imgs.asnumpy())
        # Show the losses
        loss_D, loss_G = metric[0] / metric[2], metric[1] / metric[2]
        animator.add(epoch, (loss_D, loss_G))
    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec on {str(device)}')
```

```
def train(net_D, net_G, data_iter, num_epochs, lr, latent_dim,
          device=d2l.try_gpu()):
    loss = tf.keras.losses.BinaryCrossentropy(
        from_logits=True, reduction=tf.keras.losses.Reduction.SUM)

    for w in net_D.trainable_variables:
        w.assign(tf.random.normal(mean=0, stddev=0.02, shape=w.shape))
    for w in net_G.trainable_variables:
        w.assign(tf.random.normal(mean=0, stddev=0.02, shape=w.shape))

    optimizer_hp = {"lr": lr, "beta_1": 0.5, "beta_2": 0.999}
    optimizer_D = tf.keras.optimizers.Adam(**optimizer_hp)
    optimizer_G = tf.keras.optimizers.Adam(**optimizer_hp)

    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)

    for epoch in range(1, num_epochs + 1):
```

(continues on next page)

(continued from previous page)

```
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3) # loss_D, loss_G, num_examples
        for X, _ in data_iter:
            batch_size = X.shape[0]
            Z = tf.random.normal(mean=0, stddev=1,
                                 shape=(batch_size, 1, 1, latent_dim))
            metric.add(d2l.update_D(X, Z, net_D, net_G, loss, optimizer_D),
                       d2l.update_G(Z, net_D, net_G, loss, optimizer_G),
                       batch_size)

        # Show generated examples
        Z = tf.random.normal(mean=0, stddev=1, shape=(21, 1, 1, latent_dim))
        # Normalize the synthetic data to N(0, 1)
        fake_x = net_G(Z) / 2 + 0.5
        imgs = tf.concat([tf.concat([fake_x[i * 7 + j] for j in range(7)],
                                    axis=1)
                          for i in range(len(fake_x) // 7)], axis=0)
        animator.axes[1].cla()
        animator.axes[1].imshow(imgs)
        # Show the losses
        loss_D, loss_G = metric[0] / metric[2], metric[1] / metric[2]
        animator.add(epoch, (loss_D, loss_G))
    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec on {str(device._device_
↪name)}')
```

ここでは、デモのために少ないエポック数でモデルを学習する。より良い性能を得るには、変数 num_epochs をより大きな値に設定できる。

```
latent_dim, lr, num_epochs = 100, 0.005, 20
train(net_D, net_G, data_iter, num_epochs, lr, latent_dim)
```

```
loss_D 0.296, loss_G 6.400, 2411.4 examples/sec on cuda:0
```

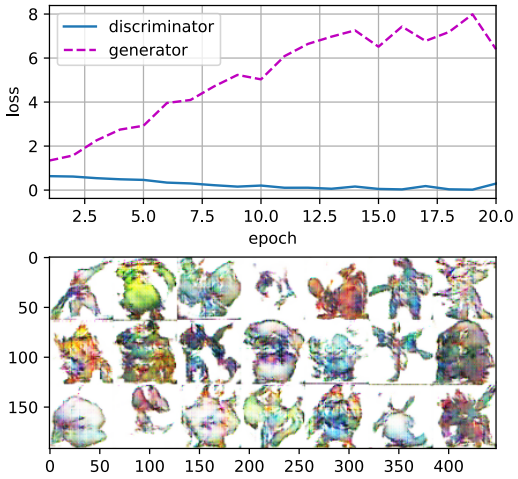
```
latent_dim, lr, num_epochs = 100, 0.005, 20
train(net_D, net_G, data_iter, num_epochs, lr, latent_dim)
```

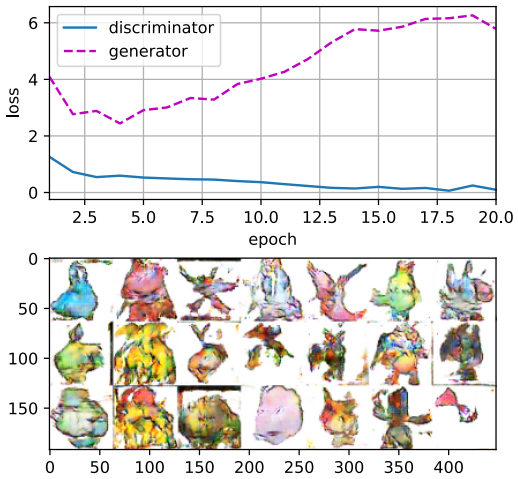
```
loss_D 0.098, loss_G 5.782, 2591.0 examples/sec on gpu(0)
```

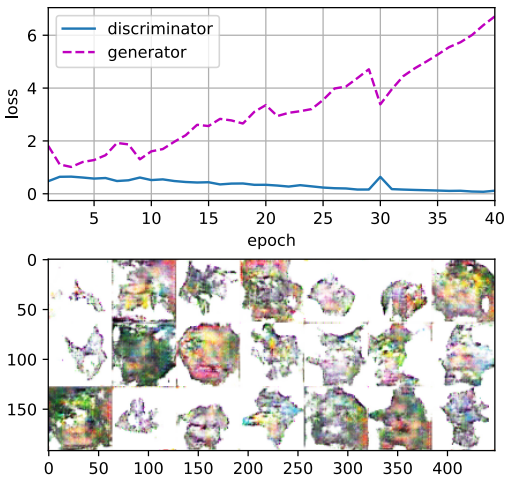
```
latent_dim, lr, num_epochs = 100, 0.0005, 40
train(net_D, net_G, data_iter, num_epochs, lr, latent_dim)
```

```
loss_D 0.116, loss_G 6.713, 2115.0 examples/sec on /GPU:0
```

23.2.5 まとめ


discriminator
generator
loss
epoch


discriminator
generator
loss
epoch


discriminator
generator
loss
epoch

- DCGAN アーキテクチャは、識別器に 4 層の畳み込み層、生成器に 4 層の「fractionally-strided」畳み込み層を持つ。
- 識別器は、バッチ正規化（入力層を除く）と leaky ReLU 活性化を備えた 4 層のストライド付き畳み込みからなる。
- Leaky ReLU は、負の入力に対して 0 でない出力を返す非線形関数である。「dying ReLU」問題を解決し、アーキテクチャ全体で勾配が流れやすくなるのを助ける。

23.2.6 演習

1. leaky ReLU の代わりに標準的な ReLU 活性化を使うとどうなるか？
2. DCGAN を Fashion-MNIST に適用し、どのカテゴリがうまくいき、どのカテゴリがうまくいかないかを調べなさい。

24 レコメンデーションシステム

Shuai Zhang (*Amazon*), **Aston Zhang** (*Amazon*), and **Yi Tay** (*Google*)

レコメンデーションシステムは産業界で広く利用されており、私たちの日常生活の至るところに存在している。これらのシステムは、オンラインショッピングサイト（例：amazon.com）、音楽・映画配信サービスサイト（例：Netflix や Spotify）、モバイルアプリストア（例：iOS app store や google play）、オンライン広告など、さまざまな分野で活用されている。

レコメンデーションシステムの主な目的は、ユーザーが閲覧すべき映画、読むべき文章、購入すべき製品といった関連性の高いアイテムを発見できるようにし、満足度の高いユーザー体験を生み出すことである。さらに、レコメンデーションシステムは、オンライン小売業者が増分収益を生み出すために導入する、最も強力な機械学習システムの一つでもある。レコメンデーションシステムは、能動的な検索にかかる手間を減らし、ユーザーが自分では検索しなかったような提案を示すことで、検索エンジンの代替にもなる。多くの企業は、より効果的なレコメンデーションシステムの助けによって競合他社より優位に立つことに成功してきた。このように、レコメンデーションシステムは私たちの日常生活にとって中心的な存在であるだけでなく、いくつかの産業では不可欠なものでもある。

この章では、レコメンデーションシステムの基礎と発展を扱うとともに、利用可能なさまざまなデータソースを用いてレコメンデーションシステムを構築するための一般的な基本技術とその実装についても見ていきる。具体的には、ユーザーがある候補アイテムに対して付けるであろう評価を予測する方法、アイテムの推薦リストを生成する方法、そして豊富な特徴量からクリック率を予測する方法を学ぶ。これらのタスクは実世界のアプリケーションでごく一般的である。この章を学ぶことで、古典的手法だけでなく、より高度な深層学習ベースのモデルを用いて、実世界の推薦問題を解くための実践的な経験を得ることができる。

24.1 レコメンダーシステムの概要

過去 10 年で、インターネットは大規模なオンラインサービスのためのプラットフォームへと進化し、私たちのコミュニケーション、ニュースの閲覧、商品の購入、映画の視聴の仕方を大きく変えた。その一方で、オンラインで提供され

る前例のない数のアイテム（ここでは *item* という用語を、映画、ニュース、書籍、製品を指すために用いる）は、私たちが好むアイテムを見つけるのを助けるシステムを必要としている。したがって、レコメンダーシステムは、パーソナライズされたサービスを実現し、個々のユーザーに合わせた体験を提供できる強力な情報フィルタリングツールである。要するに、レコメンダーシステムは、利用可能な豊富なデータを活用して選択を扱いやすくするうえで中心的な役割を果たす。今日では、レコメンダーシステムは Amazon、Netflix、YouTube など多くのオンラインサービス提供者の中核を成している。Fig. 4.3.3 における Amazon が推薦した Deep learning の書籍の例を思い出そう。レコメンダーシステムを用いる利点は二つある。第一に、ユーザーがアイテムを探す手間を大幅に減らし、情報過多の問題を緩和できる。第二に、オンラインサービス提供者にビジネス価値をもたらし、重要な収益源となる。本章では、レコメンダーシステム分野における基本概念、古典的モデル、そして深層学習による最近の進展を、実装例とともに紹介する。

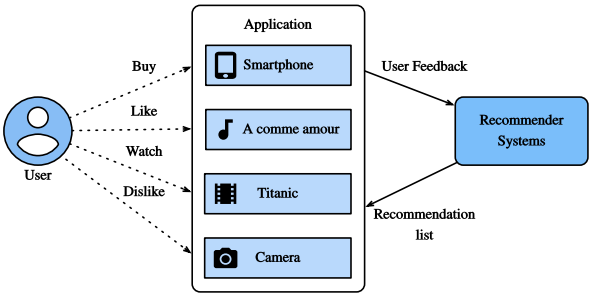


Fig. 24.1.1 推薦プロセスの図解

24.1.1 協調フィルタリング

まず、レコメンダーシステムにおける重要な概念である協調フィルタリング (CF) から始める。これは Tapestry システム (Goldberg *et al.*, 1992) によって最初に提唱され、「大量の電子メールやニュースグループに投稿されたメッセージを扱うために、人々が互いに協力してフィルタリング作業を助け合うこと」を指す。この用語は、より多くの意味を持つようになった。広い意味では、複数のユーザー、エージェント、データソース間の協力を伴う技術を用いて、情報やパターンをフィルタリングするプロセスである。CF には多くの形態があり、その登場以来、数多くの CF 手法が提案されてきた。

全体として、CF 技術は、メモリベース CF、モデルベース CF、およびそのハイブリッドに分類できる (Su and Khoshgoftaar, 2009)。代表的なメモリベース CF 技術には、ユーザーベース CF やアイテムベース CF などの最近傍ベース CF がある (Sarwar *et al.*, 2001)。行列分解のような潜在因子モデルは、モデルベース CF の例である。メモリベース CF は、共通のアイテムに基づいて類似度を計算するため、疎で大規模なデータを扱う際に限界がある。モデルベース手法は、疎性とスケーラビリティへの対応能力が高いため、より一般的になっている。多くのモデルベース CF 手法はニューラルネットワークで拡張でき、深層学習による

計算高速化とともに、より柔軟でスケーラブルなモデルにつながります (Zhang *et al.*, 2019)。一般に、CF は予測や推薦を行うためにユーザー-アイテム相互作用データのみを使用する。CF に加えて、コンテンツベースおよびコンテキストベースのレコメンダーシステムも、アイテム/ユーザーの内容記述や、タイムスタンプや位置情報のような文脈信号を取り込むうえで有用である。明らかに、利用可能な入力データが異なれば、モデルの種類や構造を調整する必要がある。

24.1.2 明示的フィードバックと暗黙的フィードバック

ユーザーの嗜好を学習するために、システムはユーザーからフィードバックを収集しなければならない。フィードバックは明示的なものにも暗黙的なものにもなりえる (Hu *et al.*, 2008)。たとえば、IMDb[133] は映画に対して 1 つ星から 10 つ星までの星評価を収集している。YouTube は、ユーザーが嗜好を示すための高評価・低評価ボタンを提供している。明示的フィードバックを集めるには、ユーザーが自ら進んで関心を示す必要があることは明らかである。とはいえ、多くのユーザーは製品を評価することに消極的であるため、明示的フィードバックが常に容易に得られるとは限らない。これに対して、暗黙的フィードバックは主としてユーザーのクリックのような暗黙的行動のモデリングに関係するため、しばしば容易に利用できる。そのため、多くのレコメンダーシステムは暗黙的フィードバックを中心に構築されており、ユーザーの行動を観察することで間接的に意見を反映する。暗黙的フィードバックには、購入履歴、閲覧履歴、視聴履歴、さらにはマウスの動きまで、さまざまな形がある。たとえば、同じ著者の本を何冊も購入したユーザーは、その著者が好きである可能性が高いだろう。暗黙的フィードバックは本質的にノイズが多いことに注意しよう。私たちにできるのは、彼らの嗜好や真の動機を 推測することだけである。あるユーザーが映画を視聴したからといって、その映画を肯定的に評価しているとは限らない。

133



24.1.3 推薦タスク

過去数十年にわたり、多くの推薦タスクが研究されてきた。応用分野に基づけば、映画推薦、ニュース推薦、POI（Point-of-Interest）推薦 (Ye *et al.*, 2011) などがある。また、フィードバックや入力データの種類に基づいてタスクを区別することもできる。たとえば、評価予測タスクは明示的な評価を予測することを目的とする。Top-*n* 推薦（アイテムランキング）は、暗黙的フィードバックに基づいて、各ユーザーごとにすべてのアイテムを個別に順位付けする。タイムスタンプ情報も含まれる場合、系列を考慮した推薦を構築できる (Quadrana *et al.*, 2018)。もう一つの人気のあるタスクはクリック率予測と呼ばれ、これも暗黙的フィードバックに基づくが、さまざまなカテゴリ特徴を利用できる。新規ユーザーへの推薦や、既存ユーザーへの新規アイテム推薦は、コールドスタート推薦と呼ばれる (Schein *et al.*, 2002)。

24.1.4 まとめ

- レコメンダーシステムは、個々のユーザーにとっても産業界にとっても重要である。協調フィルタリングは推薦における重要な概念である。
- フィードバックには、暗黙的フィードバックと明示的フィードバックの 2 種類がある。過去 10 年で、多くの推薦タスクが研究されてきた。

24.1.5 演習

1. レコメンダーシステムがあなたの日常生活にどのような影響を与えているか説明できるか？
2. どのような興味深い推薦タスクが研究できると思うか？

24.2 MovieLens データセット

134


推薦研究に利用できるデータセットはいくつもある。その中でも、MovieLens[134] データセットはおそらく最もよく知られたものの一つだろう。MovieLens は、非営利の Web ベースの映画推薦システムである。これは 1997 年に作成され、ミネソタ大学の研究室である GroupLens によって、研究目的で映画の評価データを収集するために運用されている。MovieLens のデータは、個人化推薦や社会心理学を含む多くの研究において重要な役割を果たしてきた。

24.2.1 データの取得

135


MovieLens データセットは GroupLens[135] の Web サイトで公開されている。いくつかの版があるが、ここでは MovieLens 100K データセットを使用する (Herlocker *et al.*, 1999)。このデータセットは、943 人のユーザによる 1682 本の映画に対する 100,000 件の評価から構成され、評価値は 1 から 5 までの星評価である。各ユーザが少なくとも 20 本の映画を評価するように整備されている。ユーザやアイテムに関する年齢、性別、ジャンルなどの簡単な属性情報も利用できる。

136


ml-100k.zip[136] をダウンロードして、`u.data` ファイルを展開する。このファイルには、100,000 件すべての評価が csv 形式で含まれている。フォルダ内には他にも多くのファイルがあり、各ファイルの詳細な説明はデータセットの README[137] ファイルにある。

137


まず、この節の実験を実行するために必要なパッケージをインポートしよう。

```
import os
import pandas as pd
from mxnet import gluon, np
from d2l import mxnet as d2l
```

次に、MovieLens 100k データセットをダウンロードし、相互作用を DataFrame として読み込みる。

```
#@save
d2l.DATA_HUB['ml-100k'] = (
    'https://files.grouplens.org/datasets/movielens/ml-100k.zip',
    'cd4dcac4241c8a4ad7badc7ca635da8a69dddb83')

#@save
def read_data_ml100k():
    data_dir = d2l.download_extract('ml-100k')
    names = ['user_id', 'item_id', 'rating', 'timestamp']
    data = pd.read_csv(os.path.join(data_dir, 'u.data'), sep='\t',
                       names=names, engine='python')
    num_users = data.user_id.unique().shape[0]
    num_items = data.item_id.unique().shape[0]
    return data, num_users, num_items
```

24.2.2 データセットの統計

データを読み込み、最初の 5 件のレコードを手動で確認してみよう。これはデータ構造を理解し、正しく読み込まれていることを確認するのに有効である。

```
data, num_users, num_items = read_data_ml100k()
sparsity = 1 - len(data) / (num_users * num_items)
print(f'number of users: {num_users}, number of items: {num_items}')
print(f'matrix sparsity: {sparsity:f}')
print(data.head(5))
```

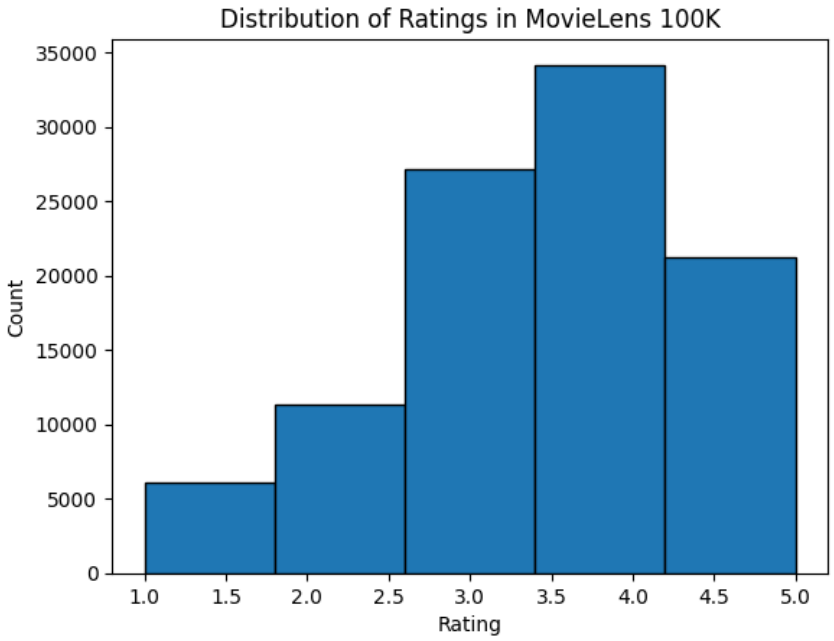
```
Downloading ../data/ml-100k.zip from https://files.grouplens.org/datasets/
↪movielens/ml-100k.zip...
number of users: 943, number of items: 1682
matrix sparsity: 0.936953
   user_id  item_id  rating  timestamp
0      196      242       3  881250949
1      186      302       3  891717742
2       22      377       1  878887116
3      244       51       2  880606923
4      166      346       1  886397596
```

各行は 4 つの列からなり、「user id」1-943、「item id」1-1682、「rating」1-5、および「timestamp」を含んでいることがわかる。ユーザ数を n、アイテム数を m とすると、サイズ $n \times m$ の相互作用行列を構成できる。このデータセットは既存の評価のみを記録しているので、評価行列と呼ぶこともできる。行列の値が正確な評価値を表す場合には、相互作用行列と評価行列を同義に扱う。評価行列の値の大部分は未知である。なぜなら、ユーザは映画の大半を評価していないからである。また、このデータセットの疎性も示している。疎性は 1 - 非ゼロ要素数 / (ユーザ数 * アイテム数) と定義される。明らかに、この相互作用行列は非常に疎である（すなわち、疎性 = 93.695%）。実世界のデータセットでは

さらに高い疎性に悩まされることがあり、推薦システム構築における長年の課題となってきた。実用的な解決策としては、ユーザ/アイテムの特徴量などの追加の側情報を用いて疎性を緩和する方法がある。

次に、さまざまな評価値の件数分布を描画する。予想どおり、これは正規分布のように見え、評価の多くは 3〜4 に集中している。

```
d2l.plt.hist(data['rating'], bins=5, ec='black')
d2l.plt.xlabel('Rating')
d2l.plt.ylabel('Count')
d2l.plt.title('Distribution of Ratings in MovieLens 100K')
d2l.plt.show()
```



24.2.3 データセットの分割

データセットを訓練セットとテストセットに分割する。以下の関数は、`random` と `seq-aware` の 2 つの分割モードを提供する。`random` モードでは、timestamp を考慮せずに 100k 件の相互作用をランダムに分割し、デフォルトではデータの 90% を訓練サンプル、残りの 10% をテストサンプルとして使用する。`seq-aware` モードでは、ユーザが最も最近に評価したアイテムをテスト用に取り除き、ユーザの過去の相互作用を訓練セットとする。ユーザの過去の相互作用は、timestamp に基づいて古いものから新しいものへと並べ替えられる。このモードは、系列を考慮した推薦の節で使用する。

```
#@save
def split_data_ml100k(data, num_users, num_items,
                      split_mode='random', test_ratio=0.1):
    """Split the dataset in random mode or seq-aware mode."""
    if split_mode == 'seq-aware':
        train_items, test_items, train_list = {}, {}, []
        for line in data.itertuples():
            u, i, rating, time = line[1], line[2], line[3], line[4]
            train_items.setdefault(u, []).append((u, i, rating, time))
            if u not in test_items or test_items[u][-1] < time:
                test_items[u] = (i, rating, time)
        for u in range(1, num_users + 1):
            train_list.extend(sorted(train_items[u], key=lambda k: k[3]))
        test_data = [(key, *value) for key, value in test_items.items()]
        train_data = [item for item in train_list if item not in test_data]
        train_data = pd.DataFrame(train_data)
        test_data = pd.DataFrame(test_data)
    else:
        mask = [True if x == 1 else False for x in np.random.uniform(
            0, 1, (len(data))) < 1 - test_ratio]
        neg_mask = [not x for x in mask]
        train_data, test_data = data[mask], data[neg_mask]
    return train_data, test_data
```

実際には、テストセットだけでなく検証セットも用いるのがよい実践である。ただし、簡潔さのためここでは省略する。この場合、テストセットはホールドアウトした検証セットとみなすことができる。

24.2.4 データの読み込み

データセットを分割した後、便宜上、訓練セットとテストセットをリストおよび辞書/行列に変換する。以下の関数は DataFrame を行ごとに読み込み、ユーザ/アイテムのインデックスを 0 から始まるように付け直す。その後、ユーザ、アイテム、評価のリストと、相互作用を記録する辞書/行列を返す。フィードバックの種類は `explicit` または `implicit` に指定できる。

```
#@save
def load_data_ml100k(data, num_users, num_items, feedback='explicit'):
    users, items, scores = [], [], []
    inter = np.zeros((num_items, num_users)) if feedback == 'explicit' else {}
    for line in data.itertuples():
        user_index, item_index = int(line[1] - 1), int(line[2] - 1)
        score = int(line[3]) if feedback == 'explicit' else 1
        users.append(user_index)
        items.append(item_index)
        scores.append(score)
        if feedback == 'implicit':
            inter.setdefault(user_index, []).append(item_index)
        else:
            inter[item_index, user_index] = score
    return users, items, scores, inter
```

その後、上記の手順をまとめて、次節で使用する。結果は `Dataset` と `Dat-`

aLoader でラップされる。なお、訓練データに対する DataLoader の last_batch は rollover モード（残りのサンプルを次のエポックに繰り越す）に設定し、順序はシャッフルする。

```
#@save
def split_and_load_ml100k(split_mode='seq-aware', feedback='explicit',
                          test_ratio=0.1, batch_size=256):
    data, num_users, num_items = read_data_ml100k()
    train_data, test_data = split_data_ml100k(
        data, num_users, num_items, split_mode, test_ratio)
    train_u, train_i, train_r, _ = load_data_ml100k(
        train_data, num_users, num_items, feedback)
    test_u, test_i, test_r, _ = load_data_ml100k(
        test_data, num_users, num_items, feedback)
    train_set = gluon.data.ArrayDataset(
        np.array(train_u), np.array(train_i), np.array(train_r))
    test_set = gluon.data.ArrayDataset(
        np.array(test_u), np.array(test_i), np.array(test_r))
    train_iter = gluon.data.DataLoader(
        train_set, shuffle=True, last_batch='rollover',
        batch_size=batch_size)
    test_iter = gluon.data.DataLoader(
        test_set, batch_size=batch_size)
    return num_users, num_items, train_iter, test_iter
```

24.2.5 まとめ

- MovieLens データセットは推薦研究で広く使われている。公開されており、無料で利用できる。
- 後続の節で利用するために、MovieLens 100k データセットをダウンロードして前処理する関数を定義した。

24.2.6 演習

- これに似た他の推薦データセットにはどのようなものがあるか？
- MovieLens についてさらに詳しく知るために、https://movielens.org/ のサイトを見てみよう。

Discussions[138]

138



24.3 行列分解

行列分解 (Koren *et al.*, 2009) は、推薦システムの文献で確立されたアルゴリズムである。行列分解モデルの最初のバージョンは、Simon Funk によって有名なブログ記事[139] で提案され、そこで彼は相互作用行列を因子分解するというアイデアを説明した。その後、2006 年に開催された Netflix コンテストによって広く

139



知られるようになった。当時、メディアストリーミングおよびビデオレンタル企業である Netflix は、推薦システムの性能を改善するためのコンテストを発表した。Netflix のベースライン、すなわち Cinematch を 10 パーセント改善できた最良のチームには、100 万米ドルの賞金が与えられるというものであった。そのため、このコンテストは推薦システム研究分野に大きな注目を集めた。その後、グランプリは BellKor's Pragmatic Chaos チーム、すなわち BellKor、Pragmatic Theory、BigChaos の合同チームによって獲得された（これらのアルゴリズムについては今は気にする必要はない）。最終スコアはアンサンブル解法（すなわち、多くのアルゴリズムの組み合わせ）の結果であったが、行列分解アルゴリズムは最終的なブレンドにおいて重要な役割を果たした。Netflix Grand Prize 解法の技術報告書 (Töscher *et al.*, 2009) では、採用されたモデルについて詳細に紹介されている。この節では、行列分解モデルの詳細とその実装について掘り下げる。

24.3.1 行列分解モデル

行列分解は協調フィルタリングモデルの一種である。具体的には、このモデルはユーザー-アイテム相互作用行列（たとえば評価行列）を 2 つの低ランク行列の積に分解し、ユーザー-アイテム相互作用の低ランク構造を捉える。

$\mathbf{R} \in \mathbb{R}^{m \times n}$ を、m 人のユーザーと n 個のアイテムからなる相互作用行列とし、$\mathbf{R}$ の値が明示的な評価を表すとする。ユーザー-アイテム相互作用は、ユーザー潜在行列 $\mathbf{P} \in \mathbb{R}^{m \times k}$ とアイテム潜在行列 $\mathbf{Q} \in \mathbb{R}^{n \times k}$ に分解される。ここで $k \ll m, n$ は潜在因子の次元である。$\mathbf{P}$ の u^{th} 行を $\mathbf{p}_u$、$\mathbf{Q}$ の i^{th} 行を $\mathbf{q}_i$ とする。あるアイテム i に対して、$\mathbf{q}_i$ の各要素は、そのアイテムが映画のジャンルや言語などの特徴をどの程度持っているかを表す。あるユーザー u に対して、$\mathbf{p}_u$ の各要素は、そのユーザーがアイテムの対応する特徴にどの程度関心を持っているかを表す。これらの潜在因子は、前述の例のように明白な次元を表すこともあれば、まったく解釈不能なこともある。予測評価値は次のように推定できる。

$$\hat{\mathbf{R}} = \mathbf{P}\mathbf{Q}^\top \tag{24.3.1}$$

ここで $\hat{\mathbf{R}} \in \mathbb{R}^{m \times n}$ は、$\mathbf{R}$ と同じ形状を持つ予測評価行列である。この予測規則の大きな問題の 1 つは、ユーザー/アイテムのバイアスをモデル化できないことである。たとえば、あるユーザーは高い評価を付けがちであったり、あるアイテムは品質が低いために常に低い評価を受けたりする。こうしたバイアスは実世界のアプリケーションでは一般的である。これらのバイアスを捉えるために、ユーザー固有およびアイテム固有のバイアス項が導入される。具体的には、ユーザー u がアイテム i に与える予測評価は次のように計算される。

$$\hat{\mathbf{R}}_{ui} = \mathbf{p}_u \mathbf{q}_i^\top + b_u + b_i \tag{24.3.2}$$

次に、予測評価スコアと実際の評価スコアの平均二乗誤差を最小化することで、行列分解モデルを学習する。目的関数は次のように定義される。

$$\underset{\mathbf{P},\mathbf{Q},b}{\operatorname{argmin}} \sum_{(u,i)\in\mathcal{K}} \|\mathbf{R}_{ui} - \hat{\mathbf{R}}_{ui}\|^2 + \lambda(\|\mathbf{P}\|_F^2 + \|\mathbf{Q}\|_F^2 + b_u^2 + b_i^2) \tag{24.3.3}$$

ここで λ は正則化率を表す。正則化項 $\lambda(\|\mathbf{P}\|_F^2 + \|\mathbf{Q}\|_F^2 + b_u^2 + b_i^2)$ は、パラメータ

の大きさにペナルティを課すことで過学習を防ぐために用いられる。$\mathbf{R}_{ui}$ が既知である (u, i) の組は、集合 $\mathcal{K} = \{(u, i) \mid \mathbf{R}_{ui} \text{ is known}\}$ に格納される。モデルパラメータは、確率的勾配降下法や Adam などの最適化アルゴリズムで学習できる。

行列分解モデルの直感的な図を以下に示す。

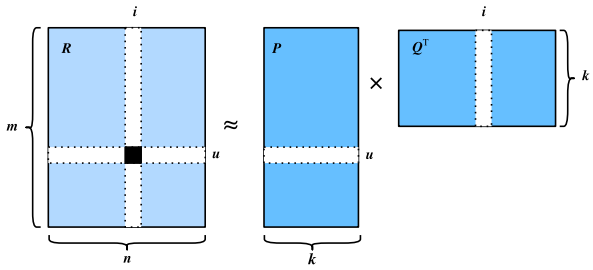


Fig. 24.3.1 Illustration of matrix factorization model

この節の残りでは、行列分解の実装を説明し、MovieLens データセットでモデルを学習する。

```
import mxnet as mx
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

24.3.2 モデルの実装

まず、上で説明した行列分解モデルを実装する。ユーザーとアイテムの潜在因子は `nn.Embedding` で作成できる。`input_dim` はアイテム/ユーザーの数であり、`output_dim` は潜在因子 k の次元である。`output_dim` を 1 に設定することで、`nn.Embedding` を使ってユーザー/アイテムのバイアスも作成できる。`forward` 関数では、ユーザー ID とアイテム ID を使って埋め込みを参照する。

```
class MF(nn.Block):
    def __init__(self, num_factors, num_users, num_items, **kwargs):
        super(MF, self).__init__(**kwargs)
        self.P = nn.Embedding(input_dim=num_users, output_dim=num_factors)
        self.Q = nn.Embedding(input_dim=num_items, output_dim=num_factors)
        self.user_bias = nn.Embedding(num_users, 1)
        self.item_bias = nn.Embedding(num_items, 1)

    def forward(self, user_id, item_id):
        P_u = self.P(user_id)
        Q_i = self.Q(item_id)
        b_u = self.user_bias(user_id)
        b_i = self.item_bias(item_id)
```

(continues on next page)

(continued from previous page)

```
        outputs = (P_u * Q_i).sum(axis=1) + np.squeeze(b_u) + np.squeeze(b_i)
        return outputs.flatten()
```

24.3.3 評価指標

次に、RMSE（root-mean-square error、二乗平均平方根誤差）を実装する。これは、モデルが予測した評価値と実際に観測された評価値（ground truth）との差を測るために一般的に用いられます (Gunawardana and Shani, 2015)。RMSE は次のように定義される。

$$\text{RMSE} = \sqrt{\frac{1}{|\mathcal{T}|} \sum_{(u,i) \in \mathcal{T}} (\mathbf{R}_{ui} - \hat{\mathbf{R}}_{ui})^2} \tag{24.3.4}$$

ここで $\mathcal{T}$ は、評価対象とするユーザーとアイテムの組からなる集合である。$|\mathcal{T}|$ はこの集合の大きさである。`mx.metric` が提供する RMSE 関数を使うことができる。

```
def evaluator(net, test_iter, devices):
    rmse = mx.metric.RMSE()  # Get the RMSE
    rmse_list = []
    for idx, (users, items, ratings) in enumerate(test_iter):
        u = gluon.utils.split_and_load(users, devices, even_split=False)
        i = gluon.utils.split_and_load(items, devices, even_split=False)
        r_ui = gluon.utils.split_and_load(ratings, devices, even_split=False)
        r_hat = [net(u, i) for u, i in zip(u, i)]
        rmse.update(labels=r_ui, preds=r_hat)
        rmse_list.append(rmse.get()[1])
    return float(np.mean(np.array(rmse_list)))
```

24.3.4 モデルの学習と評価

学習関数では、重み減衰付きの ℓ_2 損失を採用する。重み減衰の仕組みは、ℓ_2 正則化と同じ効果を持つ。

```
#@save
def train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                        devices=d2l.try_all_gpus(), evaluator=None,
                        **kwargs):
    timer = d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 2],
                            legend=['train loss', 'test RMSE'])
    for epoch in range(num_epochs):
        metric, l = d2l.Accumulator(3), 0.
        for i, values in enumerate(train_iter):
            timer.start()
            input_data = []
            values = values if isinstance(values, list) else [values]
            for v in values:
                input_data.append(gluon.utils.split_and_load(v, devices))
```

(continues on next page)

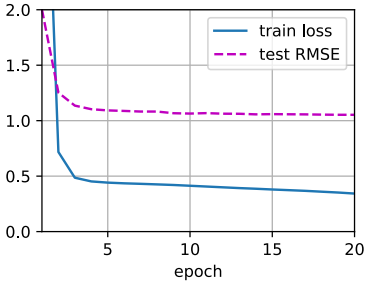
(continued from previous page)

```
            train_feat = input_data[:-1] if len(values) > 1 else input_data
            train_label = input_data[-1]
            with autograd.record():
                preds = [net(*t) for t in zip(*train_feat)]
                ls = [loss(p, s) for p, s in zip(preds, train_label)]
            [l.backward() for l in ls]
            l += sum([l.asnumpy() for l in ls]).mean() / len(devices)
            trainer.step(values[0].shape[0])
            metric.add(l, values[0].shape[0], values[0].size)
            timer.stop()
        if len(kwargs) > 0:  # It will be used in section AutoRec
            test_rmse = evaluator(net, test_iter, kwargs['inter_mat'],
                                  devices)
        else:
            test_rmse = evaluator(net, test_iter, devices)
        train_l = l / (i + 1)
        animator.add(epoch + 1, (train_l, test_rmse))
    print(f'train loss {metric[0] / metric[1]:.3f}, '
          f'test RMSE {test_rmse:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(devices)}')
```

最後に、すべてをまとめてモデルを学習しよう。ここでは、潜在因子の次元を 30 に設定する。

```
devices = d2l.try_all_gpus()
num_users, num_items, train_iter, test_iter = d2l.split_and_load_ml100k(
    test_ratio=0.1, batch_size=512)
net = MF(30, num_users, num_items)
net.initialize(ctx=devices, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.002, 20, 1e-5, 'adam'
loss = gluon.loss.L2Loss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                    devices, evaluator)
```

```
train loss 0.065, test RMSE 1.053
54640.6 examples/sec on [gpu(0), gpu(1)]
```



以下では、学習済みモデルを使って、あるユーザー（ID 20）があるアイテム（ID 30）に付けるかもしれない評価を予測する。

```
scores = net(np.array([20], dtype='int', ctx=devices[0]),
             np.array([30], dtype='int', ctx=devices[0]))
scores
```

```
array([3.272006], ctx=gpu(0))
```

24.3.5 要約

- 行列分解モデルは推薦システムで広く使われている。ユーザーがアイテムに付ける評価を予測するために利用できる。
- 推薦システム向けの行列分解を実装し、学習できる。

24.3.6 演習

- 潜在因子の大きさを変えてみよう。潜在因子の大きさはモデル性能にどのように影響するか？
- 異なる最適化手法、学習率、重み減衰率を試してみよう。
- 特定の映画に対する他のユーザーの予測評価スコアを確認してみよう。

Discussions[140]

140



24.4 AutoRec: オートエンコーダによるレーティング予測

行列分解モデルはレーティング予測タスクでまずまずの性能を達成するが、本質的には線形モデルである。そのため、このようなモデルでは、ユーザの嗜好を予測するうえで有用となりうる複雑で非線形かつ入り組んだ関係を捉えることはできない。本節では、非線形ニューラルネットワークによる協調フィルタリングモデルである AutoRec (Sedhain *et al.*, 2015) を紹介する。これは、オートエンコーダアーキテクチャによって協調フィルタリング（CF）を定式化し、明示的フィードバックに基づいて CF に非線形変換を組み込むことを目指している。ニューラルネットワークは任意の連続関数を近似できることが証明されており、行列分解の限界を補い、その表現力を高めるのに適している。

一方で、AutoRec は入力層、隠れ層、再構成（出力）層からなるオートエンコーダと同じ構造を持つ。オートエンコーダは、入力を隠れた（通常は低次元の）表現へ符号化するために、入力を出力へコピーすることを学習するニューラルネットワークである。AutoRec では、ユーザ/アイテムを明示的に低次元空間へ

埋め込む代わりに、相互作用行列の列/行を入力として用い、出力層で相互作用行列を再構成する。

他方で、AutoRec は従来のオートエンコーダとは異なる。隠れ表現を学習するのではなく、AutoRec は出力層の学習/再構成に重点を置きる。部分的に観測された相互作用行列を入力として用い、完成されたレーティング行列を再構成することを目指す。同時に、入力の欠損要素は推薦のために再構成を通じて出力層で補完される。

AutoRec には、ユーザベースとアイテムベースの 2 つの変種がある。簡潔さのため、ここではアイテムベースの AutoRec のみを紹介する。ユーザベースの AutoRec は同様に導出できる。

24.4.1 モデル

$\mathbf{R}_{*i}$ をレーティング行列の i^{th} 列とし、未知のレーティングはデフォルトで 0 に設定されているとする。ニューラルアーキテクチャは次のように定義される。

$$h(\mathbf{R}_{*i}) = f(\mathbf{W} \cdot g(\mathbf{V}\mathbf{R}_{*i} + \mu) + b) \tag{24.4.1}$$

ここで $f(\cdot)$ と $g(\cdot)$ は活性化関数、$\mathbf{W}$ と $\mathbf{V}$ は重み行列、μ と b はバイアスである。$h(\cdot)$ を AutoRec のネットワーク全体を表すものとする。出力 $h(\mathbf{R}_{*i})$ はレーティング行列の i^{th} 列の再構成である。

次の目的関数は再構成誤差の最小化を目指す。

$$\underset{\mathbf{W},\mathbf{V},\mu,b}{\operatorname{argmin}} \sum_{i=1}^{M} \| \mathbf{R}_{*i} - h(\mathbf{R}_{*i}) \|_O^2 + \lambda(\|\mathbf{W}\|_F^2 + \|\mathbf{V}\|_F^2) \tag{24.4.2}$$

ここで $\|\cdot\|_O$ は観測されたレーティングの寄与のみを考慮することを意味し、すなわち、逆伝播の際には観測された入力に対応する重みのみが更新される。

```
import mxnet as mx
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

24.4.2 モデルの実装

典型的なオートエンコーダはエンコーダとデコーダから構成される。エンコーダは入力を隠れ表現へ写像し、デコーダは隠れ層を再構成層へ写像する。ここでもこの方法に従い、全結合層を用いてエンコーダとデコーダを作成する。エンコーダの活性化関数はデフォルトで `sigmoid` に設定し、デコーダには活性化関数を適用しない。過学習を抑えるため、エンコード変換の後にドロップアウ

トを入れる。観測されていない入力の勾配はマスクし、観測されたレーティングのみがモデル学習過程に寄与するようにする。

```
class AutoRec(nn.Block):
    def __init__(self, num_hidden, num_users, dropout=0.05):
        super(AutoRec, self).__init__()
        self.encoder = nn.Dense(num_hidden, activation='sigmoid',
                                use_bias=True)
        self.decoder = nn.Dense(num_users, use_bias=True)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input):
        hidden = self.dropout(self.encoder(input))
        pred = self.decoder(hidden)
        if autograd.is_training():  # Mask the gradient during training
            return pred * np.sign(input)
        else:
            return pred
```

24.4.3 評価器の再実装

入力と出力が変更されたため、評価関数を再実装する必要があるが、精度指標としては引き続き RMSE を用いる。

```
def evaluator(network, inter_matrix, test_data, devices):
    scores = []
    for values in inter_matrix:
        feat = gluon.utils.split_and_load(values, devices, even_split=False)
        scores.extend([network(i).asnumpy() for i in feat])
    recons = np.array([item for sublist in scores for item in sublist])
    # Calculate the test RMSE
    rmse = np.sqrt(np.sum(np.square(test_data - np.sign(test_data) * recons))
                   / np.sum(np.sign(test_data)))
    return float(rmse)
```

24.4.4 モデルの学習と評価

それでは、MovieLens データセットで AutoRec を学習・評価してみよう。テスト RMSE が行列分解モデルよりも低いことがはっきりと分かり、レーティング予測タスクにおけるニューラルネットワークの有効性が確認できる。

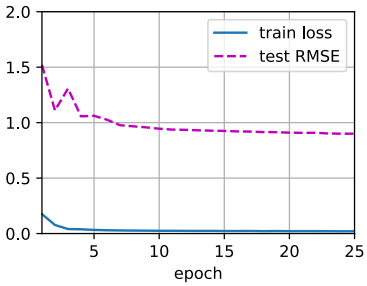
```
devices = d2l.try_all_gpus()
# Load the MovieLens 100K dataset
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items)
_, _, _, train_inter_mat = d2l.load_data_ml100k(train_data, num_users,
                                                num_items)
_, _, _, test_inter_mat = d2l.load_data_ml100k(test_data, num_users,
                                               num_items)
train_iter = gluon.data.DataLoader(train_inter_mat, shuffle=True,
                                   last_batch="rollover", batch_size=256,
```

(continues on next page)

(continued from previous page)

```
                                    num_workers=d2l.get_dataloader_workers())
test_iter = gluon.data.DataLoader(np.array(train_inter_mat), shuffle=False,
                                  last_batch="keep", batch_size=1024,
                                  num_workers=d2l.get_dataloader_workers())
# Model initialization, training, and evaluation
net = AutoRec(500, num_users)
net.initialize(ctx=devices, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.002, 25, 1e-5, 'adam'
loss = gluon.loss.L2Loss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
d2l.train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                        devices, evaluator, inter_mat=test_inter_mat)
```

```
train loss 0.000, test RMSE 0.901
10157381.7 examples/sec on [gpu(0), gpu(1)]
```



24.4.5 まとめ

- オートエンコーダを用いて行列分解アルゴリズムを定式化しつつ、非線形層とドロップアウト正則化を組み込むことができる。
- MovieLens 100K データセットでの実験により、AutoRec は行列分解よりも優れた性能を達成することが示される。

24.4.6 演習

- AutoRec の隠れ次元を変化させ、モデル性能への影響を確認しよ。
- 隠れ層をさらに追加してみよ。モデル性能の改善に役立ちますか？
- デコーダとエンコーダの活性化関数のより良い組み合わせを見つけられますか？

141



Discussions[141]

24.5 レコメンダシステムのための個人化ランキング

前の節では、明示的フィードバックのみを考慮し、モデルは観測された評価値で学習・テストされていた。このような手法には 2 つの欠点がある。第 1 に、現実世界のシナリオではフィードバックの大半は明示的ではなく暗黙的であり、明示的フィードバックの収集にはより高いコストがかかることである。第 2 に、ユーザーの興味を予測するうえで有用である可能性のある、未観測のユーザー・アイテム対が完全に無視されてしまうため、評価値がランダムに欠損しているのではなく、ユーザーの嗜好によって欠損している場合には、これらの手法は適していない。未観測のユーザー・アイテム対は、実際の負のフィードバック（ユーザーがそのアイテムに興味を持っていない）と欠損値（将来そのアイテムと相互作用するかもしれない）の混合である。行列分解と AutoRec では、未観測の対を単純に無視していた。明らかに、これらのモデルは観測済み対と未観測対を区別できず、通常、個人化ランキングタスクには適していない。

このため、暗黙的フィードバックから順位付けされた推薦リストを生成することを目的とした推薦モデル群が人気を集めている。一般に、個人化ランキングモデルは pointwise、pairwise、listwise のいずれかのアプローチで最適化できる。Pointwise アプローチは一度に 1 つの相互作用を考慮し、分類器または回帰器を学習して個々の嗜好を予測する。行列分解と AutoRec は pointwise の目的関数で最適化される。Pairwise アプローチは各ユーザーについてアイテムのペアを考慮し、そのペアに対する最適な順序を近似することを目指す。通常、pairwise アプローチは、相対的な順序を予測することがランキングの本質に近いため、ランキングタスクにより適している。Listwise アプローチは、アイテム全体のリストの順序を近似する。たとえば、Normalized Discounted Cumulative Gain (NDCG [142]) のようなランキング指標を直接最適化する。ただし、listwise アプローチは pointwise や pairwise よりも複雑で計算コストが高くなる。この節では、2 つの pairwise 目的関数／損失である Bayesian Personalized Ranking loss と Hinge loss、およびそれぞれの実装を紹介する。

142



24.5.1 Bayesian Personalized Ranking Loss とその実装

Bayesian personalized ranking (BPR) (Rendle *et al.*, 2009) は、最大事後推定量から導かれる pairwise の個人化ランキング損失である。これは既存の多くの推薦モデルで広く用いられている。BPR の学習データは、正例と負例（欠損値）の両方のペアから構成される。これは、ユーザーが正例アイテムを他のすべての未観測アイテムより好むと仮定する。

形式的には、学習データは (u, i, j) の形のタプルで構成され、これはユーザー u がアイテム i をアイテム j より好むことを表す。事後確率を最大化することを目

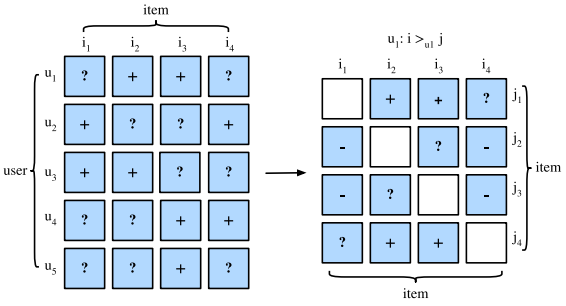
的とする BPR のベイズ的定式化は次のように与えられる。

$$p(\Theta \mid >_u) \propto p(>_u \mid \Theta) p(\Theta) \tag{24.5.1}$$

ここで Θ は任意の推薦モデルのパラメータを表し、$>_u$ はユーザー u に対する全アイテムの望ましい個人化された全順序を表す。最大事後推定量を定式化することで、個人化ランキングタスクの一般的な最適化基準を導ける。

$$\begin{aligned}
\text{BPR-OPT} : &= \ln p(\Theta \mid >_u) \\
&\propto \ln p(>_u \mid \Theta) p(\Theta) \\
&= \ln \prod_{(u,i,j \in D)} \sigma(\hat{y}_{ui} - \hat{y}_{uj}) p(\Theta) \\
&= \sum_{(u,i,j \in D)} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) + \ln p(\Theta) \\
&= \sum_{(u,i,j \in D)} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) - \lambda_\Theta \|\Theta\|^2
\end{aligned} \tag{24.5.2}$$

ここで $D \stackrel{\text{def}}{=} \{(u,i,j) \mid i \in I_u^+ \wedge j \in I \backslash I_u^+\}$ は学習集合であり、I_u^+ はユーザー u が好んだアイテム、I は全アイテム、$I \backslash I_u^+$ はユーザーが好んだアイテムを除くその他すべてのアイテムを表す。$\hat{y}_{ui}$ と $\hat{y}_{uj}$ は、それぞれユーザー u に対するアイテム i と j の予測スコアである。事前分布 $p(\Theta)$ は、平均 0、分散共分散行列 Σ_Θ をもつ正規分布である。ここでは $\Sigma_\Theta = \lambda_\Theta I$ とする。



基底クラス `mxnet.gluon.loss.Loss` を実装し、`forward` メソッドをオーバーライドして Bayesian personalized ranking loss を構成する。まず、Loss クラスと np モジュールをインポートする。

BPR loss の実装は次のとおりである。

24.5.2 Hinge Loss とその実装

143



ランキングのための Hinge loss は、SVM などの分類器でよく使われる gluon ライブラリ提供の hinge loss[143] とは異なる形をしている。レコメンダシステムにおけるランキング用の損失は、次の形をとる。

$$\sum_{(u,i,j \in D)} \max(m - \hat{y}_{ui} + \hat{y}_{uj}, 0) \tag{24.5.3}$$

ここで m は安全マージンの大きさである。これは、負のアイテムを正のアイテムから遠ざけることを目的としている。BPR と同様に、絶対的な出力ではなく正例と負例の相対的な距離を最適化しようとするため、レコメンダシステムに非常に適している。

これら 2 つの損失は、推薦における個人化ランキングで互換的に使用できる。

24.5.3 まとめ

- レコメンダシステムにおける個人化ランキングタスクには、pointwise、pairwise、listwise の 3 種類のランキング損失がある。
- 2 つの pairwise 損失である Bayesian personalized ranking loss と hinge loss は、互換的に使用できる。

24.5.4 演習

- BPR と hinge loss には、利用可能な変種はあるか？
- BPR または hinge loss を使用する推薦モデルを見つけられますか？

24.6 パーソナライズドランキングのためのニューラル協調フィルタリング

この節では明示的フィードバックを超えて、暗黙的フィードバックを用いた推薦のためのニューラル協調フィルタリング（NCF）フレームワークを導入する。暗黙的フィードバックは推薦システムに広く存在する。クリック、購入、視聴といった行動は、収集が容易でユーザの嗜好を示す一般的な暗黙的フィードバックである。ここで紹介する NeuMF (He *et al.*, 2017) は neural matrix factorization の略で、暗黙的フィードバックを用いたパーソナライズドランキング課題に取り組むことを目的としている。このモデルは、行列分解の内積をニューラルネットワークの柔軟性と非線形性で置き換えることで、モデル表現力の向上を狙っている。具体的には、このモデルは generalized matrix factorization（GMF）と MLP を含む 2 つのサブネットワークで構成され、単純な内積ではなく 2 つの経路から相互作用をモデル化する。これら 2 つのネットワークの出力は連結され、最終的な予測スコアの計算に用いられる。AutoRec のレーティング予測課題とは異なり、このモデルは暗黙的フィードバックに基づいて各ユーザに対する順位付き推薦リストを生成する。モデルの学習には、前節で導入したパーソナライズドランキング損失を用いる。

24.6.1 NeuMF モデル

前述のとおり、NeuMF は 2 つのサブネットワークを融合する。GMF は行列分解の一般的なニューラルネットワーク版であり、入力はユーザとアイテムの潜在因子の要素ごとの積である。これは 2 つのニューラル層から成る。

$$\begin{aligned} \mathbf{x} &= \mathbf{p}_u \odot \mathbf{q}_i \\ \hat{y}_{ui} &= \alpha(\mathbf{h}^\top \mathbf{x}), \end{aligned} \tag{24.6.1}$$

ここで $\odot$ はベクトルのハダマード積を表す。$\mathbf{P} \in \mathbb{R}^{m \times k}$ と $\mathbf{Q} \in \mathbb{R}^{n \times k}$ は、それぞれユーザとアイテムの潜在行列に対応する。$\mathbf{p}_u \in \mathbb{R}^k$ は P の u^{th} 行、$\mathbf{q}_i \in \mathbb{R}^k$ は Q の i^{th} 行である。α と h は出力層の活性化関数と重みを表す。$\hat{y}_{ui}$ はユーザ u がアイテム i に与えるであろう予測スコアである。

このモデルのもう一つの構成要素は MLP である。モデルの柔軟性を高めるため、MLP サブネットワークは GMF とユーザおよびアイテムの埋め込みを共有しない。代わりに、ユーザ埋め込みとアイテム埋め込みの連結を入力として用いる。複雑な接続と非線形変換により、ユーザとアイテムの間の複雑な相互作用を推定できる。より正確には、MLP サブネットワークは次のように定義される。

$$\begin{aligned} z^{(1)} &= \phi_1(\mathbf{U}_u, \mathbf{V}_i) = [\mathbf{U}_u, \mathbf{V}_i] \\ \phi^{(2)}(z^{(1)}) &= \alpha^1(\mathbf{W}^{(2)} z^{(1)} + b^{(2)}) \\ &\dots \\ \phi^{(L)}(z^{(L-1)}) &= \alpha^L(\mathbf{W}^{(L)} z^{(L-1)} + b^{(L)})) \\ \hat{y}_{ui} &= \alpha(\mathbf{h}^\top \phi^L(z^{(L-1)})) \end{aligned} \tag{24.6.2}$$

ここで $\mathbf{W}^*, \mathbf{b}^*$ および α^* は、それぞれ重み行列、バイアスベクトル、活性化関数を表す。ϕ^* は対応する層の関数を表す。$\mathbf{z}^*$ は対応する層の出力を表す。

GMF と MLP の結果を融合するために、単純な加算ではなく、NeuMF は 2 つのサブネットワークの最後から 2 番目の層を連結して、さらに後続の層に渡せる特徴ベクトルを作りる。その後、出力は行列 $\mathbf{h}$ とシグモイド活性化関数で射影される。予測層は次のように定式化される。

$$\hat{y}_{ui} = \sigma(\mathbf{h}^\top [\mathbf{x}, \phi^L(z^{(L-1)})]). \tag{24.6.3}$$

次の図は NeuMF のモデルアーキテクチャを示している。

```
import random
import mxnet as mx
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

24.6.2 モデルの実装

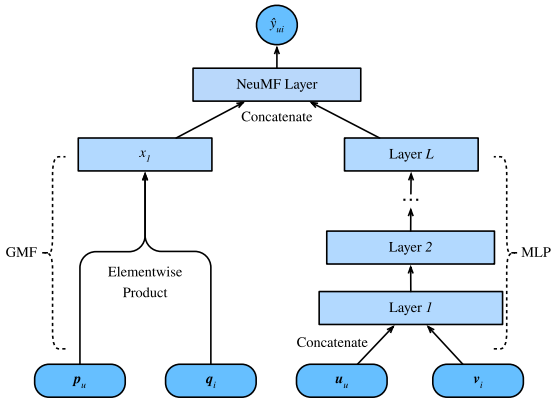


Fig. 24.6.1 Illustration of the NeuMF model

以下のコードは NeuMF モデルを実装する。これは、異なるユーザおよびアイテムの埋め込みベクトルを持つ generalized matrix factorization モデルと MLP から成る。MLP の構造はパラメータ nums_hiddens によって制御される。ReLU がデフォルトの活性化関数として使われる。

```
class NeuMF(nn.Block):
    def __init__(self, num_factors, num_users, num_items, nums_hiddens,
                 **kwargs):
        super(NeuMF, self).__init__(**kwargs)
        self.P = nn.Embedding(num_users, num_factors)
        self.Q = nn.Embedding(num_items, num_factors)
        self.U = nn.Embedding(num_users, num_factors)
        self.V = nn.Embedding(num_items, num_factors)
        self.mlp = nn.Sequential()
        for num_hiddens in nums_hiddens:
            self.mlp.add(nn.Dense(num_hiddens, activation='relu',
                                  use_bias=True))
        self.prediction_layer = nn.Dense(1, activation='sigmoid', use_
↪bias=False)

    def forward(self, user_id, item_id):
        p_mf = self.P(user_id)
        q_mf = self.Q(item_id)
        gmf = p_mf * q_mf
        p_mlp = self.U(user_id)
        q_mlp = self.V(item_id)
        mlp = self.mlp(np.concatenate([p_mlp, q_mlp], axis=1))
        con_res = np.concatenate([gmf, mlp], axis=1)
        return self.prediction_layer(con_res)
```

24.6.3 負例サンプリングを用いたカスタマイズデータセット

ペアワイズランキング損失において重要なステップは負例サンプリングである。各ユーザについて、そのユーザが相互作用していないアイテムは候補アイテム（未観測エントリ）である。次の関数はユーザ ID と候補アイテムを入力と

して受け取り、各ユーザに対して候補集合から負のアイテムをランダムにサンプリングする。学習段階では、モデルはユーザが好むアイテムが、嫌うアイテムや未相互作用のアイテムより高く順位付けされるようにする。

```
class PRDataset(gluon.data.Dataset):
    def __init__(self, users, items, candidates, num_items):
        self.users = users
        self.items = items
        self.cand = candidates
        self.all = set([i for i in range(num_items)])

    def __len__(self):
        return len(self.users)

    def __getitem__(self, idx):
        neg_items = list(self.all - set(self.cand[int(self.users[idx])]))
        indices = random.randint(0, len(neg_items) - 1)
        return self.users[idx], self.items[idx], neg_items[indices]
```

24.6.4 評価器

この節では、学習用とテスト用のデータセットを構築するために時系列分割戦略を採用する。モデルの有効性を評価するために、指定した切り捨て位置 ℓ におけるヒット率（Hit@ℓ）と ROC 曲線下面積（AUC）の 2 つの評価指標を用いる。各ユーザに対する指定位置 ℓ におけるヒット率は、推薦アイテムが上位 ℓ のランキングリストに含まれているかどうかを示す。形式的な定義は次のとおりである。

$$\text{Hit@}\ell = \frac{1}{m}\sum_{u \in \mathcal{U}} \mathbf{1}(rank_{u,g_u} <= \ell), \tag{24.6.4}$$

ここで $\mathbf{1}$ は指示関数であり、正解アイテムが上位 ℓ のリストにランクインしていれば 1、そうでなければ 0 である。$rank_{u,g_u}$ は、ユーザ u の正解アイテム g_u の推薦リスト内での順位を表す（理想的な順位は 1 です）。m はユーザ数である。$\mathcal{U}$ はユーザ集合である。

AUC の定義は次のとおりである。

$$\text{AUC} = \frac{1}{m}\sum_{u \in \mathcal{U}} \frac{1}{|\mathcal{I} \setminus S_u|} \sum_{j \in \mathcal{I} \setminus S_u} \mathbf{1}(rank_{u,g_u} < rank_{u,j}), \tag{24.6.5}$$

ここで $\mathcal{I}$ はアイテム集合である。S_u はユーザ u の候補アイテムである。なお、precision、recall、normalized discounted cumulative gain（NDCG）など、他の多くの評価プロトコルも利用できる。

次の関数は、各ユーザについてヒット数と AUC を計算する。

```
#@save
def hit_and_auc(rankedlist, test_matrix, k):
    hits_k = [(idx, val) for idx, val in enumerate(rankedlist[:k])
              if val in set(test_matrix)]
```

(continues on next page)

(continued from previous page)

```
    hits_all = [(idx, val) for idx, val in enumerate(rankedlist)
                if val in set(test_matrix)]
    max = len(rankedlist) - 1
    auc = 1.0 * (max - hits_all[0][0]) / max if len(hits_all) > 0 else 0
    return len(hits_k), auc
```

次に、全体のヒット率と AUC は次のように計算される。

```
#@save
def evaluate_ranking(net, test_input, seq, candidates, num_users, num_items,
                     devices):
    ranked_list, ranked_items, hit_rate, auc = {}, {}, [], []
    all_items = set([i for i in range(num_users)])
    for u in range(num_users):
        neg_items = list(all_items - set(candidates[int(u)]))
        user_ids, item_ids, x, scores = [], [], [], []
        [item_ids.append(i) for i in neg_items]
        [user_ids.append(u) for _ in neg_items]
        x.extend([np.array(user_ids)])
        if seq is not None:
            x.append(seq[user_ids, :])
        x.extend([np.array(item_ids)])
        test_data_iter = gluon.data.DataLoader(
            gluon.data.ArrayDataset(*x), shuffle=False, last_batch="keep",
            batch_size=1024)
        for index, values in enumerate(test_data_iter):
            x = [gluon.utils.split_and_load(v, devices, even_split=False)
                 for v in values]
            scores.extend([list(net(*t).asnumpy()) for t in zip(*x)])
        scores = [item for sublist in scores for item in sublist]
        item_scores = list(zip(item_ids, scores))
        ranked_list[u] = sorted(item_scores, key=lambda t: t[1], reverse=True)
        ranked_items[u] = [r[0] for r in ranked_list[u]]
        temp = hit_and_auc(ranked_items[u], test_input[u], 50)
        hit_rate.append(temp[0])
        auc.append(temp[1])
    return np.mean(np.array(hit_rate)), np.mean(np.array(auc))
```

24.6.5 モデルの学習と評価

学習関数を以下に定義する。モデルはペアワイズ方式で学習する。

```
#@save
def train_ranking(net, train_iter, test_iter, loss, trainer, test_seq_iter,
                  num_users, num_items, num_epochs, devices, evaluator,
                  candidates, eval_step=1):
    timer, hit_rate, auc = d2l.Timer(), 0, 0
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 1],
                            legend=['test hit rate', 'test AUC'])
    for epoch in range(num_epochs):
        metric, l = d2l.Accumulator(3), 0.
        for i, values in enumerate(train_iter):
```

(continues on next page)

(continued from previous page)

```
            input_data = []
            for v in values:
                input_data.append(gluon.utils.split_and_load(v, devices))
            with autograd.record():
                p_pos = [net(*t) for t in zip(*input_data[:-1])]
                p_neg = [net(*t) for t in zip(*input_data[:-2],
                                              input_data[-1])]
                ls = [loss(p, n) for p, n in zip(p_pos, p_neg)]
            [l.backward(retain_graph=False) for l in ls]
            l += sum([l.asnumpy() for l in ls]).mean()/len(devices)
            trainer.step(values[0].shape[0])
            metric.add(l, values[0].shape[0], values[0].size)
            timer.stop()
        with autograd.predict_mode():
            if (epoch + 1) % eval_step == 0:
                hit_rate, auc = evaluator(net, test_iter, test_seq_iter,
                                          candidates, num_users, num_items,
                                          devices)
                animator.add(epoch + 1, (hit_rate, auc))
    print(f'train loss {metric[0] / metric[1]:.3f}, '
          f'test hit rate {float(hit_rate):.3f}, test AUC {float(auc):.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(devices)}')
```

ここで、MovieLens 100k データセットを読み込み、モデルを学習できる。MovieLens データセットにはレーティングしかないため、精度の一部を失うものの、これらのレーティングを 0 と 1 に二値化する。ユーザがアイテムを評価した場合、その暗黙的フィードバックを 1 とみなし、そうでなければ 0 とする。アイテムを評価する行為は、暗黙的フィードバックを与える一形態とみなせる。ここでは、ユーザが最後に相互作用したアイテムをテスト用に残す seq-aware モードでデータセットを分割する。

```
batch_size = 1024
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items,
                                              'seq-aware')
users_train, items_train, ratings_train, candidates = d2l.load_data_ml100k(
    train_data, num_users, num_items, feedback="implicit")
users_test, items_test, ratings_test, test_iter = d2l.load_data_ml100k(
    test_data, num_users, num_items, feedback="implicit")
train_iter = gluon.data.DataLoader(
    PRDataset(users_train, items_train, candidates, num_items ), batch_size,
    True, last_batch="rollover", num_workers=d2l.get_dataloader_workers())
```

次に、モデルを作成して初期化する。隠れ層サイズが一定の 10 の 3 層 MLP を使用する。

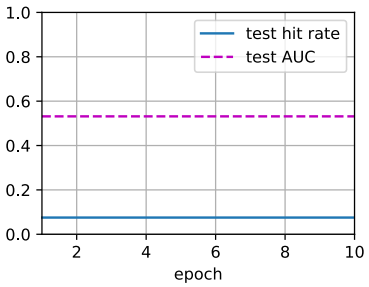
```
devices = d2l.try_all_gpus()
net = NeuMF(10, num_users, num_items, nums_hiddens=[10, 10, 10])
net.initialize(ctx=devices, force_reinit=True, init=mx.init.Normal(0.01))
```

```
[07:23:17] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
[07:23:18] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
[07:23:18] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

以下のコードでモデルを学習する。

```
lr, num_epochs, wd, optimizer = 0.01, 10, 1e-5, 'adam'
loss = d2l.BPRLoss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
train_ranking(net, train_iter, test_iter, loss, trainer, None, num_users,
              num_items, num_epochs, devices, evaluate_ranking, candidates)
```

```
train loss 16.982, test hit rate 0.075, test AUC 0.531
2.6 examples/sec on [gpu(0), gpu(1)]
```



24.6.6 まとめ

- 行列分解モデルに非線形性を加えることは、モデルの能力と有効性を改善するうえで有益である。
- NeuMF は行列分解と MLP の組み合わせである。MLP はユーザとアイテムの埋め込みの連結を入力として受け取りる。

24.6.7 演習

- 潜在因子の次元を変えてみよう。潜在因子のサイズはモデル性能にどのような影響を与えるだろうか？
- MLP のアーキテクチャ（例：層数、各層のニューロン数）を変えて、その性能への影響を確認しよう。
- 異なる最適化手法、学習率、重み減衰率を試してみよう。
- 前節で定義したヒンジ損失を使ってこのモデルを最適化してみよう。

Discussions[144]

144



24.7 シーケンスを考慮した推薦システム

前の節では、ユーザの短期的な行動を考慮せずに、推薦タスクを行列補完問題として抽象化した。この節では、時系列順に並んだユーザのインタラクションログを考慮する推薦モデルを紹介する。これはシーケンスを考慮した推薦モデル (Quadrana *et al.*, 2018) であり、入力は順序付きで、しばしばタイムスタンプ付きの過去のユーザ行動のリストである。近年の多くの研究では、このような情報を組み込むことが、ユーザの時間的な行動パターンをモデル化し、興味の変化を発見するうえで有用であることが示されている。

ここで紹介するモデル Caser (Tang and Wang, 2018) は convolutional sequence embedding recommendation model の略で、畳み込みニューラルネットワークを用いて、ユーザの最近の活動がもたらす動的なパターンの影響を捉える。Caser の主要部分は水平畳み込みネットワークと垂直畳み込みネットワークからなり、それぞれ union-level と point-level のシーケンスパターンを抽出することを目的としている。point-level パターンは履歴シーケンス中の単一アイテムが対象アイテムに与える影響を示し、union-level パターンは複数の過去の行動が後続の対象に与える影響を意味する。たとえば、牛乳とバターを一緒に買うと、そのどちらか一方だけを買う場合よりも、小麦粉を買う確率が高くなる。さらに、ユーザの一般的な興味、すなわち長期的な嗜好も最後の全結合層でモデル化され、より包括的なユーザ興味のモデリングが実現される。モデルの詳細は以下のとおりである。

24.7.1 モデルアーキテクチャ

シーケンスを考慮した推薦システムでは、各ユーザはアイテム集合からのいくつかのアイテムのシーケンスに対応付けられる。$S^u = (S_1^u, ...S_{|S_u|}^u)$ を順序付きシーケンスとする。Caser の目的は、ユーザの一般的な嗜好と短期的な意図の両方を考慮してアイテムを推薦することである。直前の L 個のアイテムを考慮すると、時刻 t におけるそれ以前のインタラクションを表す埋め込み行列は次のように構成できる。

$$\mathbf{E}^{(u,t)} = [\mathbf{q}_{S_{t-L}^u}, ..., \mathbf{q}_{S_{t-2}^u}, \mathbf{q}_{S_{t-1}^u}]^\top, \tag{24.7.1}$$

ここで $\mathbf{Q} \in \mathbb{R}^{n \times k}$ はアイテム埋め込みを表し、$\mathbf{q}_i$ はその i^{th} 行を表す。$\mathbf{E}^{(u,t)} \in \mathbb{R}^{L \times k}$ は、時刻ステップ t におけるユーザ u の一時的な興味を推定するために用いることができる。入力行列 $\mathbf{E}^{(u,t)}$ は画像のように見なすことができ、後続の 2 つの畳み込みコンポーネントへの入力となる。

水平畳み込み層は d 個の水平フィルタ $\mathbf{F}^j \in \mathbb{R}^{h \times k}, 1 \le j \le d, h = \{1, ..., L\}$ を持ち、垂直畳み込み層は d' 個の垂直フィルタ $\mathbf{G}^j \in \mathbb{R}^{L \times 1}, 1 \le j \le d'$ を持つ。一連

の畳み込みとプーリング操作の後、次の 2 つの出力が得られる。

$$\begin{aligned} \mathbf{o} &= \text{HConv}(\mathbf{E}^{(u,t)}, \mathbf{F}) \\ \mathbf{o}' &= \text{VConv}(\mathbf{E}^{(u,t)}, \mathbf{G}), \end{aligned} \tag{24.7.2}$$

ここで $\mathbf{o} \in \mathbb{R}^d$ は水平畳み込みネットワークの出力であり、$\mathbf{o}' \in \mathbb{R}^{kd'}$ は垂直畳み込みネットワークの出力である。簡単のため、畳み込みとプーリングの詳細は省略する。これらを連結し、より高次の表現を得るために全結合ニューラルネットワーク層へ入力する。

$$\mathbf{z} = \phi(\mathbf{W}[\mathbf{o}, \mathbf{o}']^\top + \mathbf{b}), \tag{24.7.3}$$

ここで $\mathbf{W} \in \mathbb{R}^{k \times (d+kd')}$ は重み行列、$\mathbf{b} \in \mathbb{R}^k$ はバイアスである。学習されたベクトル $\mathbf{z} \in \mathbb{R}^k$ は、ユーザの短期的な意図を表現したものである。

最後に、予測関数はユーザの短期的な嗜好と一般的な嗜好を組み合わせ、次のように定義される。

$$\hat{y}_{uit} = \mathbf{v}_i \cdot [\mathbf{z}, \mathbf{p}_u]^\top + \mathbf{b}'_i, \tag{24.7.4}$$

ここで $\mathbf{V} \in \mathbb{R}^{n \times 2k}$ は別のアイテム埋め込み行列である。$\mathbf{b}' \in \mathbb{R}^n$ はアイテム固有のバイアスである。$\mathbf{P} \in \mathbb{R}^{m \times k}$ は、ユーザの一般的な嗜好のためのユーザ埋め込み行列である。$\mathbf{p}_u \in \mathbb{R}^k$ は P の u^{th} 行であり、$\mathbf{v}_i \in \mathbb{R}^{2k}$ は $\mathbf{V}$ の i^{th} 行である。

このモデルは BPR 損失または Hinge 損失で学習できる。Caser のアーキテクチャを以下に示す。

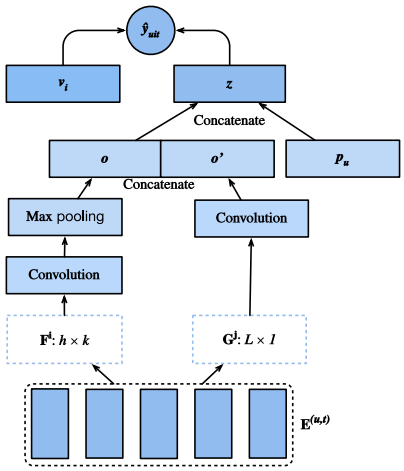


Fig. 24.7.1 Illustration of the Caser Model

まず、必要なライブラリをインポートする。

```
import random
import mxnet as mx
from mxnet import gluon, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

24.7.2 モデルの実装

以下のコードは Caser モデルを実装したものである。これは垂直畳み込み層、水平畳み込み層、および全結合層から構成される。

```
class Caser(nn.Block):
    def __init__(self, num_factors, num_users, num_items, L=5, d=16,
                 d_prime=4, drop_ratio=0.05, **kwargs):
        super(Caser, self).__init__(**kwargs)
        self.P = nn.Embedding(num_users, num_factors)
        self.Q = nn.Embedding(num_items, num_factors)
        self.d_prime, self.d = d_prime, d
        # Vertical convolution layer
        self.conv_v = nn.Conv2D(d_prime, (L, 1), in_channels=1)
        # Horizontal convolution layer
        h = [i + 1 for i in range(L)]
        self.conv_h, self.max_pool = nn.Sequential(), nn.Sequential()
        for i in h:
            self.conv_h.add(nn.Conv2D(d, (i, num_factors), in_channels=1))
            self.max_pool.add(nn.MaxPool1D(L - i + 1))
        # Fully connected layer
        self.fc1_dim_v, self.fc1_dim_h = d_prime * num_factors, d * len(h)
        self.fc = nn.Dense(in_units=d_prime * num_factors + d * L,
                           activation='relu', units=num_factors)
        self.Q_prime = nn.Embedding(num_items, num_factors * 2)
        self.b = nn.Embedding(num_items, 1)
        self.dropout = nn.Dropout(drop_ratio)

    def forward(self, user_id, seq, item_id):
        item_embs = np.expand_dims(self.Q(seq), 1)
        user_emb = self.P(user_id)
        out, out_h, out_v, out_hs = None, None, None, []
        if self.d_prime:
            out_v = self.conv_v(item_embs)
            out_v = out_v.reshape(out_v.shape[0], self.fc1_dim_v)
        if self.d:
            for conv, maxp in zip(self.conv_h, self.max_pool):
                conv_out = np.squeeze(npx.relu(conv(item_embs)), axis=3)
                t = maxp(conv_out)
                pool_out = np.squeeze(t, axis=2)
                out_hs.append(pool_out)
            out_h = np.concatenate(out_hs, axis=1)
        out = np.concatenate([out_v, out_h], axis=1)
        z = self.fc(self.dropout(out))
        x = np.concatenate([z, user_emb], axis=1)
        q_prime_i = np.squeeze(self.Q_prime(item_id))
```

(continues on next page)

(continued from previous page)

```
        b = np.squeeze(self.b(item_id))
        res = (x * q_prime_i).sum(1) + b
        return res
```

24.7.3 負例サンプリング付きのシーケンシャルデータセット

シーケンシャルなインタラクションデータを処理するには、Dataset クラスを再実装する必要がある。以下のコードは SeqDataset という新しいデータセットクラスを作成する。各サンプルでは、ユーザ ID、過去にインタラクションした L 個のアイテムをシーケンスとして、そして次にインタラクションするアイテムをターゲットとして出力する。次の図は、1 人のユーザに対するデータ読み込みの流れを示している。このユーザが 9 本の映画を気に入ったとする。これら 9 本の映画を時系列順に並べる。最新の映画はテスト用アイテムとして除外される。残りの 8 本の映画については、3 つの訓練サンプルを得ることができ、各サンプルは 5 個の映画からなるシーケンス（$L = 5$）と、その後続のアイテムをターゲットアイテムとして含む。カスタマイズしたデータセットには負例サンプルも含まれる。

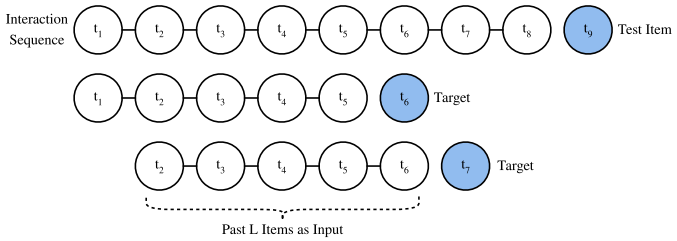


Fig. 24.7.2 Illustration of the data generation process

```
class SeqDataset(gluon.data.Dataset):
    def __init__(self, user_ids, item_ids, L, num_users, num_items,
                 candidates):
        user_ids, item_ids = np.array(user_ids), np.array(item_ids)
        sort_idx = np.array(sorted(range(len(user_ids)),
                                   key=lambda k: user_ids[k]))
        u_ids, i_ids = user_ids[sort_idx], item_ids[sort_idx]
        temp, u_ids, self.cand = {}, u_ids.asnumpy(), candidates
        self.all_items = set([i for i in range(num_items)])
        [temp.setdefault(u_ids[i], []).append(i) for i, _ in enumerate(u_ids)]
        temp = sorted(temp.items(), key=lambda x: x[0])
        u_ids = np.array([i[0] for i in temp])
        idx = np.array([i[1][0] for i in temp])
        self.ns = ns = int(sum([c - L if c >= L + 1 else 1 for c
                                in np.array([len(i[1]) for i in temp])]))
        self.seq_items = np.zeros((ns, L))
        self.seq_users = np.zeros(ns, dtype='int32')
        self.seq_tgt = np.zeros((ns, 1))
        self.test_seq = np.zeros((num_users, L))
        test_users, _uid = np.empty(num_users), None
```

(continues on next page)

(continued from previous page)

```
        for i, (uid, i_seq) in enumerate(self._seq(u_ids, i_ids, idx, L + 1)):
            if uid != _uid:
                self.test_seq[uid][:] = i_seq[-L:]
                test_users[uid], _uid = uid, uid
            self.seq_tgt[i][:] = i_seq[-1:]
            self.seq_items[i][:], self.seq_users[i] = i_seq[:L], uid

    def _win(self, tensor, window_size, step_size=1):
        if len(tensor) - window_size >= 0:
            for i in range(len(tensor), 0, - step_size):
                if i - window_size >= 0:
                    yield tensor[i - window_size:i]
                else:
                    break
        else:
            yield tensor

    def _seq(self, u_ids, i_ids, idx, max_len):
        for i in range(len(idx)):
            stop_idx = None if i >= len(idx) - 1 else int(idx[i + 1])
            for s in self._win(i_ids[int(idx[i]):stop_idx], max_len):
                yield (int(u_ids[i]), s)

    def __len__(self):
        return self.ns

    def __getitem__(self, idx):
        neg = list(self.all_items - set(self.cand[int(self.seq_users[idx])]))
        i = random.randint(0, len(neg) - 1)
        return (self.seq_users[idx], self.seq_items[idx], self.seq_tgt[idx],
                neg[i])
```

24.7.4 MovieLens 100K データセットの読み込み

その後、MovieLens 100K データセットをシーケンスを考慮したモードで読み込み・分割し、上で実装したシーケンシャルデータローダーを用いて訓練データを読み込みる。

```
TARGET_NUM, L, batch_size = 1, 5, 4096
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items,
                                              'seq-aware')
users_train, items_train, ratings_train, candidates = d2l.load_data_ml100k(
    train_data, num_users, num_items, feedback="implicit")
users_test, items_test, ratings_test, test_iter = d2l.load_data_ml100k(
    test_data, num_users, num_items, feedback="implicit")
train_seq_data = SeqDataset(users_train, items_train, L, num_users,
                            num_items, candidates)
train_iter = gluon.data.DataLoader(train_seq_data, batch_size, True,
                                   last_batch="rollover",
                                   num_workers=d2l.get_dataloader_workers())
test_seq_iter = train_seq_data.test_seq
train_seq_data[0]
```

```
[07:21:27] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(array(0, dtype=int32),
 array([241., 170., 110., 255.,   4.]),
 array([101.]),
 945)
```

上に示したのが訓練データの構造である。最初の要素はユーザ ID、次のリストはこのユーザが気に入った直近 5 個のアイテムを示し、最後の要素はその 5 個の後にこのユーザが気に入ったアイテムである。

24.7.5 モデルの訓練

それでは、モデルを訓練しよう。比較可能な結果を得るために、前節の NeuMF と同じ設定、すなわち学習率、オプティマイザ、そして k を用いる。

```
devices = d2l.try_all_gpus()
net = Caser(10, num_users, num_items, L)
net.initialize(ctx=devices, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.04, 8, 1e-5, 'adam'
loss = d2l.BPRLoss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})

# Running takes > 1h (pending fix from MXNet)
# d2l.train_ranking(net, train_iter, test_iter, loss, trainer, test_seq_iter,␣
↪num_users, num_items, num_epochs, devices, d2l.evaluate_ranking, candidates,␣
↪eval_step=1)
```

```
[07:23:02] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
[07:23:02] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for GPU
```

24.7.6 まとめ

- ユーザの短期的および長期的な興味を推定することで、ユーザが好みそうな次のアイテムをより効果的に予測できる。
- 畳み込みニューラルネットワークは、シーケンシャルなインタラクションからユーザの短期的な興味を捉えるために利用できる。

24.7.7 演習

- 水平畳み込みネットワークと垂直畳み込みネットワークの一方を取り除くアブレーション研究を行っよ。どちらのコンポーネントがより重要だろうか？

- ハイパーパラメータ L を変えてみよ。より長い履歴のインタラクションは、より高い精度をもたらすだろうか？
- 上で紹介したシーケンスを考慮した推薦タスクのほかに、session-based recommendation と呼ばれる別の種類のシーケンスを考慮した推薦タスクがある (Hidasi *et al.*, 2015)。この 2 つのタスクの違いを説明できるか？

Discussions[145]

145



24.8 特徴豊富な推薦システム

インタラクションデータは、ユーザーの嗜好や関心を示す最も基本的な指標である。これは、前に紹介したモデルにおいて重要な役割を果たす。しかし、インタラクションデータは通常きわめて疎であり、時にはノイズを含む。この問題に対処するために、アイテムの特徴、ユーザーのプロファイル、さらにはそのインタラクションがどのような文脈で発生したかといったサイド情報を推薦モデルに統合できる。これらの特徴を活用することは、特にインタラクションデータが不足している場合に、ユーザーの関心を効果的に予測できるため、推薦に有用である。そのため、推薦モデルにはこれらの特徴を扱う能力も備わっており、モデルにコンテンツ／文脈認識を持たせることが重要である。この種の推薦モデルを示すために、オンライン広告推薦におけるクリック率（CTR）について別のタスクを導入し (McMahan *et al.*, 2013)、匿名化された広告データセットを紹介する。ターゲット広告サービスは広く注目を集めており、しばしば推薦エンジンとして扱われる。ユーザーの個人的な嗜好や関心に合った広告を推薦することは、クリック率の向上にとって重要である。

デジタルマーケターは、オンライン広告を用いて顧客に広告を表示す。クリック率は、広告主が広告に対して得たクリック数を表示回数で割った指標であり、次の式で計算される割合として表される。

$$\mathrm{CTR} = \frac{\#\mathrm{Clicks}}{\#\mathrm{Impressions}} \times 100\%. \tag{24.8.1}$$

クリック率は、予測アルゴリズムの有効性を示す重要なシグナルである。クリック率予測は、ウェブサイト上の何かがクリックされる可能性を予測するタスクである。CTR 予測のモデルは、ターゲット広告システムだけでなく、一般的なアイテム（たとえば映画、ニュース、製品）の推薦システム、メールキャンペーン、さらには検索エンジンにも利用できる。また、ユーザー満足度やコンバージョン率とも密接に関連しており、現実的な期待値を設定するのに役立つため、キャンペーン目標の設定にも有用である。

```
import os
from collections import defaultdict
from mxnet import gluon, np
from d2l import mxnet as d2l
```

24.8.1 オンライン広告データセット

インターネットとモバイル技術の大きな進歩に伴い、オンライン広告は重要な収益源となり、インターネット産業の収益の大部分を生み出している。無関係な広告ではなく、ユーザーの関心を引く広告を表示することは重要であり、そうすることで、偶然訪れた訪問者を有料顧客へと転換できる。ここで紹介するデータセットはオンライン広告データセットである。これは 34 個のフィールドからなり、最初の列は広告がクリックされたかどうかを示す目的変数（1）または（0）を表す。残りの列はすべてカテゴリ特徴である。各列は、広告 ID、サイトまたはアプリケーション ID、デバイス ID、時刻、ユーザープロファイルなどを表している可能性がある。匿名化とプライバシー保護のため、特徴の実際の意味は公開されていない。

以下のコードは、データセットをサーバーからダウンロードし、ローカルのデータフォルダに保存する。

```
#@save
d2l.DATA_HUB['ctr'] = (d2l.DATA_URL + 'ctr.zip',
                       'e18327c48c8e8e5c23da714dd614e390d369843f')

data_dir = d2l.download_extract('ctr')
```

訓練セットとテストセットがあり、それぞれ 15000 件と 3000 件のサンプル／行から構成されている。

24.8.2 データセットラッパー

データ読み込みを便利にするために、CSV ファイルから広告データセットを読み込み、`DataLoader` で使用できる `CTRDataset` を実装する。

```
#@save
class CTRDataset(gluon.data.Dataset):
    def __init__(self, data_path, feat_mapper=None, defaults=None,
                 min_threshold=4, num_feat=34):
        self.NUM_FEATS, self.count, self.data = num_feat, 0, {}
        feat_cnts = defaultdict(lambda: defaultdict(int))
        self.feat_mapper, self.defaults = feat_mapper, defaults
        self.field_dims = np.zeros(self.NUM_FEATS, dtype=np.int64)
        with open(data_path) as f:
            for line in f:
                instance = {}
                values = line.rstrip('\n').split('\t')
                if len(values) != self.NUM_FEATS + 1:
                    continue
                label = np.float32([0, 0])
                label[int(values[0])] = 1
                instance['y'] = [np.float32(values[0])]
                for i in range(1, self.NUM_FEATS + 1):
                    feat_cnts[i][values[i]] += 1
                    instance.setdefault('x', []).append(values[i])
                self.data[self.count] = instance
```

(continues on next page)

(continued from previous page)

```
                self.count = self.count + 1
        if self.feat_mapper is None and self.defaults is None:
            feat_mapper = {i: {feat for feat, c in cnt.items() if c >=
                               min_threshold} for i, cnt in feat_cnts.items()}
            self.feat_mapper = {i: {feat_v: idx for idx, feat_v in␣
↪enumerate(feat_values)}
                                for i, feat_values in feat_mapper.items()}
            self.defaults = {i: len(feat_values) for i, feat_values in feat_
↪mapper.items()}
        for i, fm in self.feat_mapper.items():
            self.field_dims[i - 1] = len(fm) + 1
        self.offsets = np.array((0, *np.cumsum(self.field_dims).asnumpy()
                                 [:-1]))

    def __len__(self):
        return self.count

    def __getitem__(self, idx):
        feat = np.array([self.feat_mapper[i + 1].get(v, self.defaults[i + 1])
                         for i, v in enumerate(self.data[idx]['x'])])
        return feat + self.offsets, self.data[idx]['y']
```

次の例では、訓練データを読み込み、最初のレコードを表示する。

```
train_data = CTRDataset(os.path.join(data_dir, 'train.csv'))
train_data[0]
```

```
[07:21:14] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
(array([ 143.,  146.,  227.,  263.,  957., 1250., 1471., 1566., 1624.,
        1712., 2008., 2061., 2162., 2304., 2305., 2360., 2745., 2746.,
        2747., 2748., 2892., 2988., 3165., 3166., 3194., 3195., 3303.,
        3664., 3687., 3690., 3728., 3755., 3773., 3793.]),
 [1.0])
```

見てわかるように、34 個のフィールドはすべてカテゴリ特徴である。各値は、対応するエントリの one-hot インデックスを表す。ラベル 0 はクリックされていないことを意味する。この `CTRDataset` は、Criteo display advertising challenge dataset[146] や Avazu click-through rate prediction dataset[147] など、他のデータセットの読み込みにも使用できる。

146



147



24.8.3 まとめ

- クリック率は、広告システムや推薦システムの有効性を測るために用いられる重要な指標である。
- クリック率予測は通常、二値分類問題に変換される。目的は、与えられた特

徴に基づいて、広告／アイテムがクリックされるかどうかを予測することである。

24.8.4 演習

- 提供された `CTRDataset` を使って、Criteo と Avazu のデータセットを読み込めますか。なお、Criteo データセットには実数値特徴が含まれているため、コードを少し修正する必要があるかもしれない。

Discussions[148]

148

24.9 因子分解機

因子分解機（FM）は、Rendle (2010) によって提案された教師ありアルゴリズムで、分類、回帰、ランキングの各タスクに利用できる。登場するとすぐに注目を集め、予測や推薦を行うための人気が高く影響力のある手法となった。特に、線形回帰モデルと行列分解モデルを一般化したものである。さらに、多項式カーネルを用いたサポートベクターマシンにも似ている。線形回帰や行列分解に対する因子分解機の利点は次のとおりである。(1) χ-項の変数相互作用をモデル化できる。ここで χ は多項式の次数であり、通常は 2 に設定される。(2) 因子分解機に関連する高速な最適化アルゴリズムにより、多項式計算の時間計算量を線形計算量まで削減でき、特に高次元の疎な入力に対して非常に効率的である。これらの理由から、因子分解機は現代の広告や製品推薦で広く用いられている。以下で技術的詳細と実装を説明する。

24.9.1 2-項因子分解機

形式的には、$x \in \mathbb{R}^d$ を 1 サンプルの特徴ベクトル、y を対応するラベルとする。ラベルは実数値ラベルでも、二値分類の「クリック/非クリック」のようなクラスラベルでもかまわない。次数 2 の因子分解機のモデルは次のように定義される。

$$\hat{y}(x) = \mathbf{w}_0 + \sum_{i=1}^{d} \mathbf{w}_i x_i + \sum_{i=1}^{d} \sum_{j=i+1}^{d} \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j \tag{24.9.1}$$

ここで、$\mathbf{w}_0 \in \mathbb{R}$ は全体のバイアス、$\mathbf{w} \in \mathbb{R}^d$ は i 番目の変数の重み、$\mathbf{V} \in \mathbb{R}^{d \times k}$ は特徴埋め込みを表す。$\mathbf{v}_i$ は $\mathbf{V}$ の i^{th} 行を表し、k は潜在因子の次元数である。$\langle \cdot, \cdot \rangle$ は 2 つのベクトルの内積である。$\langle \mathbf{v}_i, \mathbf{v}_j \rangle$ は i^{th} 特徴と j^{th} 特徴の相互作用をモデル化する。特徴相互作用の中には容易に理解できるものもあり、専門家が設計できる。しかし、それ以外の多くの特徴相互作用はデータの中に隠れており、特定が困難である。そのため、特徴相互作用を自動的にモデル化することで、特徴量設計の労力を大幅に削減できる。最初の 2 項が線形回帰モデルに対応し、最後の項が行列分解モデルの拡張であることは明らかである。特徴 i がア

イテムを表し、特徴 j がユーザを表す場合、第 3 項はユーザ埋め込みとアイテム埋め込みの内積そのものである。なお、FM はより高次（次数 > 2）にも一般化できる。ただし、数値安定性が一般化性能を弱める可能性がある。

24.9.2 効率的な最適化基準

因子分解機を素朴に最適化すると、すべてのペアワイズ相互作用を計算する必要があるため、計算量は $O(kd^2)$ になる。この非効率性を解決するために、FM の第 3 項を再整理して計算コストを大幅に削減し、線形時間計算量（$O(kd)$）にできる。ペアワイズ相互作用項の変形は次のとおりである。

$$\begin{aligned}
&\sum_{i=1}^{d}\sum_{j=i+1}^{d}\langle \mathbf{v}_i, \mathbf{v}_j\rangle x_i x_j \\
&= \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d}\langle \mathbf{v}_i, \mathbf{v}_j\rangle x_i x_j - \frac{1}{2}\sum_{i=1}^{d}\langle \mathbf{v}_i, \mathbf{v}_i\rangle x_i x_i \\
&= \frac{1}{2}\Big(\sum_{i=1}^{d}\sum_{j=1}^{d}\sum_{l=1}^{k}\mathbf{v}_{i,l}\mathbf{v}_{j,l}x_i x_j - \sum_{i=1}^{d}\sum_{l=1}^{k}\mathbf{v}_{i,l}\mathbf{v}_{i,l}x_i x_i\Big) \\
&= \frac{1}{2}\sum_{l=1}^{k}\Big((\sum_{i=1}^{d}\mathbf{v}_{i,l}x_i)(\sum_{j=1}^{d}\mathbf{v}_{j,l}x_j) - \sum_{i=1}^{d}\mathbf{v}_{i,l}^2 x_i^2\Big) \\
&= \frac{1}{2}\sum_{l=1}^{k}\Big((\sum_{i=1}^{d}\mathbf{v}_{i,l}x_i)^2 - \sum_{i=1}^{d}\mathbf{v}_{i,l}^2 x_i^2\Big)
\end{aligned} \tag{24.9.2}$$

この変形により、モデルの計算量は大幅に削減される。さらに、疎な特徴では非ゼロ要素だけを計算すればよいため、全体の計算量は非ゼロ特徴数に対して線形になる。

FM モデルの学習には、回帰タスクでは MSE 損失、分類タスクではクロスエントロピー損失、ランキングタスクでは BPR 損失を使える。確率的勾配降下法や Adam などの標準的な最適化手法が利用できる。

```
import os
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

24.9.3 モデルの実装

以下のコードでは因子分解機を実装する。FM が線形回帰ブロックと効率的な特徴相互作用ブロックから構成されていることが分かりる。CTR 予測を分類タスクとして扱うため、最終スコアにシグモイド関数を適用する。

```
class FM(nn.Block):
    def __init__(self, field_dims, num_factors):
        super(FM, self).__init__()
        num_inputs = int(sum(field_dims))
        self.embedding = nn.Embedding(num_inputs, num_factors)
        self.fc = nn.Embedding(num_inputs, 1)
        self.linear_layer = nn.Dense(1, use_bias=True)

    def forward(self, x):
        square_of_sum = np.sum(self.embedding(x), axis=1) ** 2
        sum_of_square = np.sum(self.embedding(x) ** 2, axis=1)
        x = self.linear_layer(self.fc(x).sum(1)) \
            + 0.5 * (square_of_sum - sum_of_square).sum(1, keepdims=True)
        x = npx.sigmoid(x)
        return x
```

24.9.4 広告データセットの読み込み

前節の CTR データラッパーを使ってオンライン広告データセットを読み込みる。

```
batch_size = 2048
data_dir = d2l.download_extract('ctr')
train_data = d2l.CTRDataset(os.path.join(data_dir, 'train.csv'))
test_data = d2l.CTRDataset(os.path.join(data_dir, 'test.csv'),
                           feat_mapper=train_data.feat_mapper,
                           defaults=train_data.defaults)
train_iter = gluon.data.DataLoader(
    train_data, shuffle=True, last_batch='rollover', batch_size=batch_size,
    num_workers=d2l.get_dataloader_workers())
test_iter = gluon.data.DataLoader(
    test_data, shuffle=False, last_batch='rollover', batch_size=batch_size,
    num_workers=d2l.get_dataloader_workers())
```

```
Downloading ../data/ctr.zip from http://d2l-data.s3-accelerate.amazonaws.com/
↪ctr.zip...
[07:05:14] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

24.9.5 モデルの学習

その後、モデルを学習する。学習率は 0.02、埋め込みサイズはデフォルトで 20 に設定する。モデル学習には Adam 最適化手法と SigmoidBinaryCrossEntropyLoss 損失を使用する。

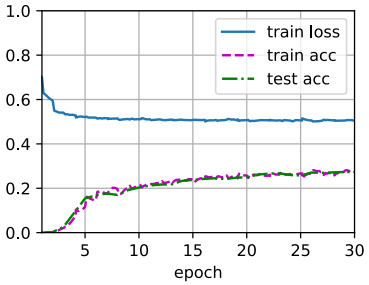
```
devices = d2l.try_all_gpus()
net = FM(train_data.field_dims, num_factors=20)
net.initialize(init.Xavier(), ctx=devices)
lr, num_epochs, optimizer = 0.02, 30, 'adam'
trainer = gluon.Trainer(net.collect_params(), optimizer,
```

(continues on next page)

(continued from previous page)

```
                        {'learning_rate': lr})
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.506, train acc 0.273, test acc 0.273
45373.6 examples/sec on [gpu(0), gpu(1)]
```



24.9.6 まとめ

- FM は、回帰、分類、ランキングなどさまざまなタスクに適用できる汎用的な枠組みである。
- 特徴相互作用/特徴交差は予測タスクにおいて重要であり、2 項の相互作用は FM により効率的にモデル化できる。

24.9.7 演習

- Avazu、MovieLens、Criteo などの他のデータセットで FM を試せますか？
- 埋め込みサイズを変えて性能への影響を確認しよ。行列分解の場合と同様の傾向が観察できるか？

Discussions[149]

149



24.10 Deep Factorization Machines

効果的な特徴の組み合わせを学習することは、クリック率予測タスクの成功にとって極めて重要である。Factorization Machines は、特徴間相互作用を線形の枠組み（たとえば双線形相互作用）でモデル化する。しかし、実世界のデータでは、特徴の交差構造は通常きわめて複雑で非線形であるため、これだけでは不十分なことが多い。さらに悪いことに、実際には通常、2 次の特徴相互作用しか factorization machines では用いられない。より高次の特徴の組み合わせを

factorization machines でモデル化することは理論上は可能だが、数値的不安定性と高い計算複雑性のため、一般には採用されない。

有効な解決策の一つは、深層ニューラルネットワークを用いることである。深層ニューラルネットワークは特徴表現学習に強力であり、洗練された特徴相互作用を学習できる可能性がある。そのため、深層ニューラルネットワークを factorization machines に統合するのは自然な発想である。factorization machines に非線形変換層を追加すると、低次の特徴の組み合わせと高次の特徴の組み合わせの両方をモデル化できるようになる。さらに、入力に由来する非線形な内在構造も深層ニューラルネットワークで捉えることができる。この節では、FM と深層ニューラルネットワークを組み合わせた代表的なモデルである deep factorization machines (DeepFM) (Guo *et al.*, 2017) を紹介する。

24.10.1 Model Architectures

DeepFM は、FM コンポーネントと deep コンポーネントからなり、これらは並列構造で統合されている。FM コンポーネントは 2-way factorization machines と同じで、低次の特徴相互作用をモデル化するために使われる。deep コンポーネントは MLP であり、高次の特徴相互作用と非線形性を捉えるために使われる。これら 2 つのコンポーネントは同じ入力／埋め込みを共有し、その出力が最終予測として加算される。DeepFM の考え方は、記憶と汎化の両方を捉えられる Wide & Deep アーキテクチャに似ていることに注意するとよいだろう。Wide & Deep モデルに対する DeepFM の利点は、特徴の組み合わせを自動的に識別することで、手作業による特徴設計の労力を減らせる点である。

簡潔さのため、FM コンポーネントの説明は省略し、その出力を $\hat{y}^{(FM)}$ と表す。詳細は前節を参照しよ。$\mathbf{e}_i \in \mathbb{R}^k$ を i^{th} フィールドの潜在特徴ベクトルとする。deep コンポーネントの入力は、疎なカテゴリ特徴入力に対してルックアップされたすべてのフィールドの密な埋め込みを連結したもので、次のように表される。

$$\mathbf{z}^{(0)} = [\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_f], \tag{24.10.1}$$

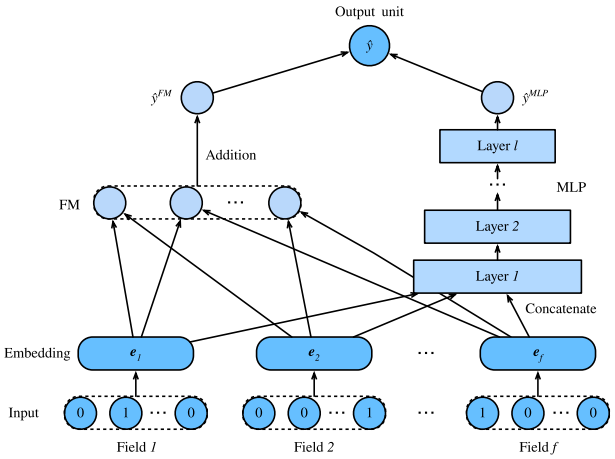
ここで f はフィールド数である。これを次のニューラルネットワークに入力する。

$$\mathbf{z}^{(l)} = \alpha(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}), \tag{24.10.2}$$

ここで α は活性化関数である。$\mathbf{W}_l$ と $\mathbf{b}_l$ は l^{th} 層の重みとバイアスである。y_{DNN} を予測出力とする。DeepFM の最終予測は、FM と DNN の両方の出力の和である。したがって、次式を得る。

$$\hat{y} = \sigma(\hat{y}^{(FM)} + \hat{y}^{(DNN)}), \tag{24.10.3}$$

ここで σ はシグモイド関数である。DeepFM のアーキテクチャを以下に示す。



DeepFM は、深層ニューラルネットワークと FM を組み合わせる唯一の方法ではないことに注意しよ。特徴相互作用の上に非線形層を追加することもできる (He and Chua, 2017)。

```
import os
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

24.10.2 Implementation of DeepFM

DeepFM の実装は FM のそれと似ている。FM 部分はそのままにし、活性化関数として relu を用いた MLP ブロックを使う。モデルの正則化のために Dropout も用いる。MLP のニューロン数は mlp_dims ハイパーパラメータで調整できる。

```
class DeepFM(nn.Block):
    def __init__(self, field_dims, num_factors, mlp_dims, drop_rate=0.1):
        super(DeepFM, self).__init__()
        num_inputs = int(sum(field_dims))
        self.embedding = nn.Embedding(num_inputs, num_factors)
        self.fc = nn.Embedding(num_inputs, 1)
        self.linear_layer = nn.Dense(1, use_bias=True)
        input_dim = self.embed_output_dim = len(field_dims) * num_factors
        self.mlp = nn.Sequential()
        for dim in mlp_dims:
            self.mlp.add(nn.Dense(dim, 'relu', True, in_units=input_dim))
            self.mlp.add(nn.Dropout(rate=drop_rate))
            input_dim = dim
        self.mlp.add(nn.Dense(in_units=input_dim, units=1))
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        embed_x = self.embedding(x)
        square_of_sum = np.sum(embed_x, axis=1) ** 2
        sum_of_square = np.sum(embed_x ** 2, axis=1)
        inputs = np.reshape(embed_x, (-1, self.embed_output_dim))
        x = self.linear_layer(self.fc(x).sum(1)) \
            + 0.5 * (square_of_sum - sum_of_square).sum(1, keepdims=True) \
            + self.mlp(inputs)
        x = npx.sigmoid(x)
        return x
```

24.10.3 Training and Evaluating the Model

データ読み込みの手順は FM と同じである。DeepFM の MLP コンポーネントは、ピラミッド構造（30-20-10）を持つ 3 層の全結合ネットワークに設定する。その他のハイパーパラメータは FM と同じである。

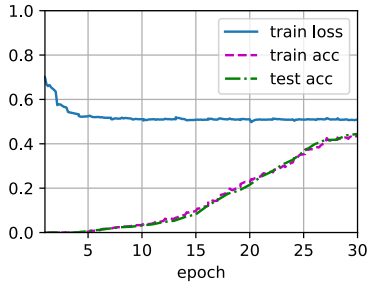
```
batch_size = 2048
data_dir = d2l.download_extract('ctr')
train_data = d2l.CTRDataset(os.path.join(data_dir, 'train.csv'))
test_data = d2l.CTRDataset(os.path.join(data_dir, 'test.csv'),
                           feat_mapper=train_data.feat_mapper,
                           defaults=train_data.defaults)
field_dims = train_data.field_dims
train_iter = gluon.data.DataLoader(
    train_data, shuffle=True, last_batch='rollover', batch_size=batch_size,
    num_workers=d2l.get_dataloader_workers())
test_iter = gluon.data.DataLoader(
    test_data, shuffle=False, last_batch='rollover', batch_size=batch_size,
    num_workers=d2l.get_dataloader_workers())
devices = d2l.try_all_gpus()
net = DeepFM(field_dims, num_factors=10, mlp_dims=[30, 20, 10])
net.initialize(init.Xavier(), ctx=devices)
lr, num_epochs, optimizer = 0.01, 30, 'adam'
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {'learning_rate': lr})
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.509, train acc 0.437, test acc 0.444
42903.5 examples/sec on [gpu(0), gpu(1)]
```

FM と比べると、DeepFM はより速く収束し、より良い性能を達成する。

24.10.4 Summary

- FM にニューラルネットワークを統合することで、複雑で高次の相互作用をモデル化できるようになる。
- DeepFM は広告データセットにおいて、元の FM を上回る性能を示す。



24.10.5 Exercises

- MLP の構造を変えて、モデル性能への影響を確認しよ。
- データセットを Criteo に変更し、元の FM モデルと比較しよ。

Discussions[150]

150



25 付録: 深層学習のための数学

Brent Werness (*Amazon*), **Rachel Hu** (*Amazon*), および本書の著者たち

現代の深層学習の素晴らしい点の一つは、その多くが、背後にある数学を完全に理解していなくても理解し、利用できることである。これは、この分野が成熟しつつあることの表れである。ほとんどのソフトウェア開発者がもはや計算可能関数の理論を気にする必要がないのと同じように、深層学習の実践者も最尤学習の理論的基礎を気にする必要はないはずである。

しかし、まだそこまでは至っていない。

実際には、アーキテクチャ上の選択が勾配の流れにどのような影響を与えるのか、あるいは特定の損失関数で学習することで暗黙にどのような仮定を置いているのかを理解する必要があることがある。エントロピーが一体何を測っているのか、そしてそれがモデルにおける bits-per-character の意味を正確に理解するのにどう役立つのかを知る必要があるかもしれない。これらはいずれも、より深い数学的理解を必要とする。

この付録は、現代の深層学習の中核理論を理解するために必要な数学的背景を提供することを目的としているが、網羅的ではない。まず、線形代数をより深く見ていきる。一般的な線形代数の対象と操作について幾何学的な理解を構築し、それによってさまざまな変換がデータに与える影響を視覚化できるようにする。重要な要素として、固有分解の基礎を扱う。

次に、勾配が最急降下の方向である理由、そして逆伝播がなぜそのような形を取るのかを完全に理解できるところまで、微分積分学の理論を発展させる。その後、次の話題である確率論を支えるのに必要な程度まで、積分積分学について議論する。

実際に遭遇する問題はしばしば確定的ではないため、不確実なものについて語るための言語が必要である。そこで、確率変数の理論と、最もよく遭遇する分布を概観し、モデルを確率的に議論できるようにする。これにより、確率的分類手法であるナイーブベイズ分類器の基礎が得られる。

確率論と密接に関連しているのが統計学である。統計学は短い節で扱うにはあまりにも広大な分野であるが、すべての機械学習実践者が知っておくべき基本概念、特に推定量の評価と比較、仮説検定の実施、信頼区間の構成を紹介する。

最後に、情報の保存と伝達を数学的に研究する情報理論を扱う。これにより、ある対象領域についてモデルがどれだけの情報を保持しているかを定量的に議論するための中核的な言語が得られる。

これらを合わせると、深層学習を深く理解する道へ進み始めるために必要な数学的概念の中核を成する。

25.1 幾何と線形代数的操作

Section 5.3 では、線形代数の基礎に触れ、それを用いてデータを変換する一般的な操作を表現できることを見た。線形代数は、深層学習や、より広く機械学習で行う多くの仕事を支える重要な数学的柱の一つである。Section 5.3 には現代の深層学習モデルの仕組みを伝えるのに十分な道具立てがあったが、この分野にはまだまだ多くの内容がある。この節では、さらに深く掘り下げ、線形代数の操作に関するいくつかの幾何学的解釈を示し、固有値と固有ベクトルを含むいくつかの基本概念を導入する。

25.1.1 ベクトルの幾何

まず、ベクトルの 2 つの一般的な幾何学的解釈、すなわち空間内の点または方向としての解釈について議論する必要がある。本質的には、ベクトルとは下の Python のリストのような数の並びである。

```
v = [1, 7, 0, 1]
```

数学者はこれをたいてい 列ベクトルまたは 行ベクトルとして書く。つまり、次のように表する。

$$\mathbf{x} = \begin{bmatrix} 1 \\ 7 \\ 0 \\ 1 \end{bmatrix}, \tag{25.1.1}$$

あるいは

$$\mathbf{x}^\top = \begin{bmatrix} 1 & 7 & 0 & 1 \end{bmatrix}. \tag{25.1.2}$$

これらはしばしば異なる解釈を持ち、データ例は列ベクトル、重み付き和を作るために使う重みは行ベクトルとして扱われる。しかし、柔軟であることは有益である。Section 5.3 で述べたように、単一のベクトルの既定の向きは列ベクトルであるが、表形式データセットを表す任意の行列については、各データ例を行ベクトルとして行列内で扱う方がより一般的である。

ベクトルが与えられたとき、まず与えるべき解釈は空間内の点としての解釈である。2 次元または 3 次元では、固定された基準である 原点に対して、ベクトルの成分を用いて空間内の点の位置を定めることで、これらの点を可視化できる。これは Fig. 25.1.1 に示されている。

この幾何学的な見方により、問題をより抽象的なレベルで考えられるように

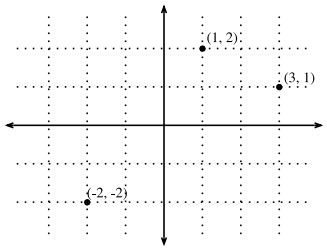


Fig. 25.1.1 平面上の点としてベクトルを可視化する図。ベクトルの第 1 成分は x 座標を、第 2 成分は y 座標を与える。高次元でも同様だが、可視化ははるかに難しい。

なる。猫か犬かを分類するような、いかにも手に負えなさそうな問題に直面する代わりに、課題を空間内の点の集合として抽象的に捉え、2 つの異なる点のクラスターをどう分離するかを見つけることとして考え始めることができる。

並行して、ベクトルに対して人々がしばしば取る第 2 の見方がある。それは空間内の方向としての見方である。ベクトル $\mathbf{v} = [3, 2]^\top$ を、原点から右に 3、上に 2 の位置として考えるだけでなく、右に 3 歩、上に 2 歩進むための方向そのものとしても考えられる。このようにして、図 Fig. 25.1.2 にあるすべてのベクトルを同じものとして扱う。

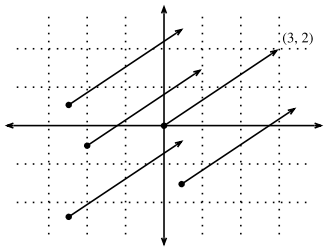


Fig. 25.1.2 任意のベクトルは平面上の矢印として可視化できる。この場合、描かれているすべてのベクトルはベクトル $(3, 2)^\top$ の表現である。

この見方の利点の一つは、ベクトル加算の操作を視覚的に理解できることである。特に、まず一方のベクトルが示す方向に進み、その後で他方のベクトルが示す方向に進む、ということを行いる。これは Fig. 25.1.3 に示されている。

ベクトル減算にも同様の解釈がある。$\mathbf{u} = \mathbf{v} + (\mathbf{u} - \mathbf{v})$ という恒等式を考えると、ベクトル $\mathbf{u} - \mathbf{v}$ は点 $\mathbf{v}$ から点 $\mathbf{u}$ へ移動する方向であることがわかる。

25.1.2 内積と角度

Section 5.3 で見たように、2 つの列ベクトル $\mathbf{u}$ と $\mathbf{v}$ を取ると、次のように計算してその内積を作れる。

$$\mathbf{u}^\top \mathbf{v} = \sum_i u_i \cdot v_i. \tag{25.1.3}$$

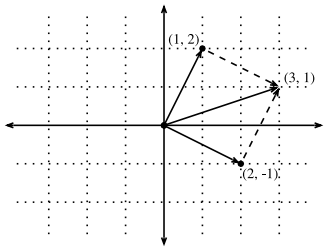


Fig. 25.1.3 まず一方のベクトルに従い、その後で別のベクトルに従うことで、ベクトル加算を可視化できる。

(25.1.3) は対称的なので、古典的な掛け算の記法にならって

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^\top \mathbf{v} = \mathbf{v}^\top \mathbf{u}, \tag{25.1.4}$$

と書き、ベクトルの順序を入れ替えても同じ答えになることを強調する。

内積 (25.1.3) には幾何学的な解釈もある。それは 2 つのベクトルのなす角と密接に関係している。Fig. 25.1.4 に示す角を考えよう。

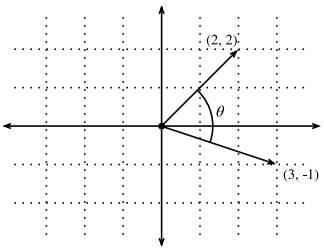


Fig. 25.1.4 平面内の任意の 2 つのベクトルの間には、定義された角 θ がある。この角は内積と密接に結びついている。

まず、2 つの特定のベクトルを考える。

$$\mathbf{v} = (r, 0) \text{ and } \mathbf{w} = (s\cos(\theta), s\sin(\theta)). \tag{25.1.5}$$

ベクトル $\mathbf{v}$ は長さ r で x 軸に平行に伸びており、ベクトル $\mathbf{w}$ は長さ s で x 軸となす角が θ である。この 2 つのベクトルの内積を計算すると、

$$\mathbf{v} \cdot \mathbf{w} = rs\cos(\theta) = \|\mathbf{v}\|\|\mathbf{w}\|\cos(\theta). \tag{25.1.6}$$

簡単な代数操作により、項を整理して

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{25.1.7}$$

を得られる。

要するに、この 2 つの特定のベクトルについては、内積とノルムを組み合わせることで 2 つのベクトルのなす角がわかる。この事実は一般の場合にも成り立つ。ここでは導出はしないが、$\|\mathbf{v} - \mathbf{w}\|^2$ を 2 通りに書くことを考えると、一方

は内積を用い、もう一方は余弦定理を用いることで、完全な関係を得られる。実際、任意の 2 つのベクトル $\mathbf{v}$ と $\mathbf{w}$ について、そのなす角は

$$\theta = \arccos\left(\frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{25.1.8}$$

である。

この結果は、計算のどこにも 2 次元という情報が現れないので、うれしい性質である。実際、3 次元でも 300 万次元でも問題なく使える。

簡単な例として、ベクトルの組のなす角をどう計算するか見てみよう。

```
%matplotlib inline
from d2l import torch as d2l
from IPython import display
import torch
from torchvision import transforms
import torchvision

def angle(v, w):
    return torch.acos(v.dot(w) / (torch.norm(v) * torch.norm(w)))

angle(torch.tensor([0, 1, 2], dtype=torch.float32), torch.tensor([2.0, 3, 4]))
```

```
tensor(0.4190)
```

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from mxnet import gluon, np, npx
npx.set_np()

def angle(v, w):
    return np.arccos(v.dot(w) / (np.linalg.norm(v) * np.linalg.norm(w)))

angle(np.array([0, 1, 2]), np.array([2, 3, 4]))
```

```
[07:04:33] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
array(0.41899002)
```

```
%matplotlib inline
from d2l import tensorflow as d2l
from IPython import display
import tensorflow as tf

def angle(v, w):
    return tf.acos(tf.tensordot(v, w, axes=1) / (tf.norm(v) * tf.norm(w)))
```

(continues on next page)

(continued from previous page)

```
angle(tf.constant([0, 1, 2], dtype=tf.float32), tf.constant([2.0, 3, 4]))
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=0.41899002>
```

今は使わないが、角度が $\pi/2$（同値に 90°）であるベクトルを 直交していると言うことを知っておくと便利である。上の式を見ると、これは $\theta = \pi/2$ のときに起こり、これは $\cos(\theta) = 0$ と同じことである。これが起こる唯一の方法は内積そのものが 0 である場合であり、2 つのベクトルは $\mathbf{v} \cdot \mathbf{w} = 0$ のとき、かつそのときに限り直交する。これは、対象を幾何学的に理解するときに役立つ式である。

なぜ角度を計算することが有用なのか、という疑問はもっともである。答えは、データに期待する不変性の種類にある。画像と、その複製で各ピクセル値が同じだが明るさが 10% のものを考えよう。個々のピクセルの値は、一般に元の値からかなり離れている。したがって、元の画像と暗い画像の距離を計算すると、その距離は大きくなりえる。しかし、多くの機械学習アプリケーションでは、内容は同じである。—猫/犬分類器にとっては、依然として猫の画像である。一方、角度を考えると、任意のベクトル $\mathbf{v}$ について、$\mathbf{v}$ と $0.1 \cdot \mathbf{v}$ のなす角は 0 であることはすぐにわかる。これは、ベクトルをスケーリングしても向きは同じで、長さだけが変わることに対応している。角度は、暗い画像を同一とみなする。

このような例は至るところにある。テキストでは、同じことを言っている文書を 2 倍の長さで書いたとしても、議論している話題が変わらないことを望むかもしれない。ある種の符号化（たとえば、ある語彙における単語の出現回数を数えるもの）では、これは文書を表すベクトルの 2 倍化に対応するので、やはり角度を使える。

コサイン類似度

2 つのベクトルの近さを測るために角度を用いる機械学習の文脈では、実務家はコサイン類似度という用語を使って、次の量を指する。

$$\cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}. \tag{25.1.9}$$

コサインは、2 つのベクトルが同じ方向を向くときに最大値 1、反対方向を向くときに最小値 –1、2 つのベクトルが直交するときに値 0 を取りる。高次元ベクトルの成分が平均 0 でランダムにサンプルされる場合、そのコサインはほとんど常に 0 に近くなることに注意する。

25.1.3 超平面

ベクトルを扱うことに加えて、線形代数を深く理解するためにもう一つ重要な対象がある。それは 超平面であり、これは線（2 次元）や平面（3 次元）の高次

元への一般化である。d 次元ベクトル空間では、超平面は $d-1$ 次元を持ち、空間を 2 つの半空間に分割する。

例から始めよう。列ベクトル $\mathbf{w} = [2, 1]^\top$ があるとする。「$\mathbf{w} \cdot \mathbf{v} = 1$ を満たす点 $\mathbf{v}$ はどれか？」を知りたいとする。上で述べた内積と角度の関係 (25.1.8) を思い出すと、これは次と同値である。

$$\|\mathbf{v}\|\|\mathbf{w}\| \cos(\theta) = 1 \iff \|\mathbf{v}\| \cos(\theta) = \frac{1}{\|\mathbf{w}\|} = \frac{1}{\sqrt{5}}. \tag{25.1.10}$$

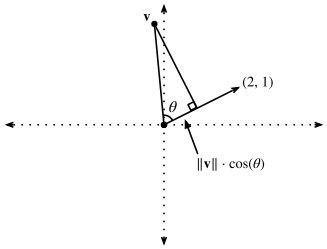


Fig. 25.1.5 三角法を思い出すと、式 $\|\mathbf{v}\| \cos(\theta)$ はベクトル $\mathbf{v}$ を $\mathbf{w}$ の方向へ射影した長さである。

この式の幾何学的意味を考えると、これは $\mathbf{v}$ を $\mathbf{w}$ の方向へ射影した長さがちょうど $1/\|\mathbf{w}\|$ である、と言っているのと同じだとわかる。これは Fig. 25.1.5 に示されている。これが成り立つ点の集合は、ベクトル $\mathbf{w}$ に直角な直線である。必要なら、この直線の方程式を求めて、$2x + y = 1$、あるいは同値に $y = 1 - 2x$ であることを確認できる。

次に、$\mathbf{w} \cdot \mathbf{v} > 1$ や $\mathbf{w} \cdot \mathbf{v} < 1$ を満たす点の集合を考えると、それぞれ射影が $1/\|\mathbf{w}\|$ より長い、または短い場合であることがわかる。したがって、これら 2 つの不等式は直線の両側を定める。このようにして、空間を 2 つに切り分ける方法が得られる。一方の側のすべての点は内積がしきい値未満で、もう一方の側はしきい値超過である。これは Fig. 25.1.6 に示されている。

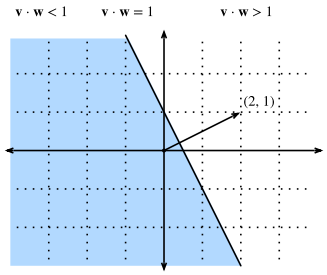


Fig. 25.1.6 式の不等式版を考えると、超平面（この場合は単なる直線）が空間を 2 つに分けることがわかる。

高次元でも話はほぼ同じである。今度は $\mathbf{w} = [1, 2, 3]^\top$ を取り、3 次元で $\mathbf{w} \cdot \mathbf{v} = 1$

を満たす点を考えると、与えられたベクトル $\mathbf{w}$ に直角な平面が得られる。2 つの不等式は、再び平面の両側を定める。これは Fig. 25.1.7 に示されている。

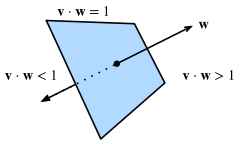


Fig. 25.1.7 任意の次元の超平面は空間を 2 つに分ける。

ここで可視化できる能力は尽きるが、10 次元でも、100 次元でも、10 億次元でもこれを行うことを妨げるものはない。これは機械学習モデルを考えるときによく起こりる。たとえば、Section 7.1 のような線形分類モデルは、異なる目標クラスを分離する超平面を見つける方法として理解できる。この文脈では、そのような超平面はしばしば 決定平面と呼ばれる。深層学習による分類モデルの大半は、最後に softmax に入力される線形層で終わるので、深層ニューラルネットワークの役割は、目標クラスが超平面によってきれいに分離できるような非線形埋め込みを見つけることだと解釈できる。

手作りの例として、Fashion-MNIST データセット（Section 7.2 で見たもの）に含まれる T シャツとズボンの小さな画像を分類するために、それらの平均の差ベクトルを使って決定平面を定め、大まかなしきい値を目分量で決めるだけでも、かなり妥当なモデルを作れることに注意する。まずデータを読み込み、平均を計算する。

```
# Load in the dataset
trans = []
trans.append(transforms.ToTensor())
trans = transforms.Compose(trans)
train = torchvision.datasets.FashionMNIST(root="../data", transform=trans,
                                          train=True, download=True)
test = torchvision.datasets.FashionMNIST(root="../data", transform=trans,
                                         train=False, download=True)

X_train_0 = torch.stack(
    [x[0] * 256 for x in train if x[1] == 0]).type(torch.float32)
X_train_1 = torch.stack(
    [x[0] * 256 for x in train if x[1] == 1]).type(torch.float32)
X_test = torch.stack(
    [x[0] * 256 for x in test if x[1] == 0 or x[1] == 1]).type(torch.float32)
y_test = torch.stack([torch.tensor(x[1]) for x in test
                      if x[1] == 0 or x[1] == 1]).type(torch.float32)

# Compute averages
ave_0 = torch.mean(X_train_0, axis=0)
ave_1 = torch.mean(X_train_1, axis=0)
```

```
# Load in the dataset
train = gluon.data.vision.FashionMNIST(train=True)
test = gluon.data.vision.FashionMNIST(train=False)
```

(continues on next page)

(continued from previous page)

```
X_train_0 = np.stack([x[0] for x in train if x[1] == 0]).astype(float)
X_train_1 = np.stack([x[0] for x in train if x[1] == 1]).astype(float)
X_test = np.stack(
    [x[0] for x in test if x[1] == 0 or x[1] == 1]).astype(float)
y_test = np.stack(
    [x[1] for x in test if x[1] == 0 or x[1] == 1]).astype(float)

# Compute averages
ave_0 = np.mean(X_train_0, axis=0)
ave_1 = np.mean(X_train_1, axis=0)
```

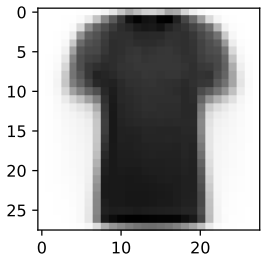
```
# Load in the dataset
((train_images, train_labels), (
    test_images, test_labels)) = tf.keras.datasets.fashion_mnist.load_data()


X_train_0 = tf.cast(tf.stack(train_images[[i for i, label in enumerate(
    train_labels) if label == 0]] * 256), dtype=tf.float32)
X_train_1 = tf.cast(tf.stack(train_images[[i for i, label in enumerate(
    train_labels) if label == 1]] * 256), dtype=tf.float32)
X_test = tf.cast(tf.stack(test_images[[i for i, label in enumerate(
    test_labels) if label == 0]] * 256), dtype=tf.float32)
y_test = tf.cast(tf.stack(test_images[[i for i, label in enumerate(
    test_labels) if label == 1]] * 256), dtype=tf.float32)

# Compute averages
ave_0 = tf.reduce_mean(X_train_0, axis=0)
ave_1 = tf.reduce_mean(X_train_1, axis=0)
```

これらの平均を詳しく調べると有益なので、どのように見えるかを描画してみよう。この場合、平均は確かにぼやけた T シャツの画像に見える。

```
# Plot average t-shirt
d2l.set_figsize()
d2l.plt.imshow(ave_0.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```

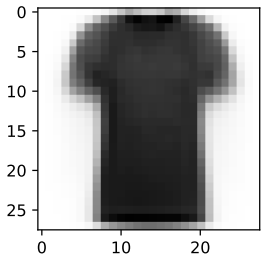



```
# Plot average t-shirt
```

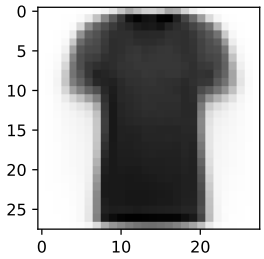
(continues on next page)

(continued from previous page)

```
d2l.set_figsize()
d2l.plt.imshow(ave_0.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```

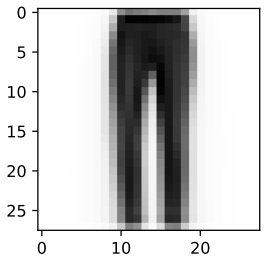



```
# Plot average t-shirt
d2l.set_figsize()
d2l.plt.imshow(tf.reshape(ave_0, (28, 28)), cmap='Greys')
d2l.plt.show()
```

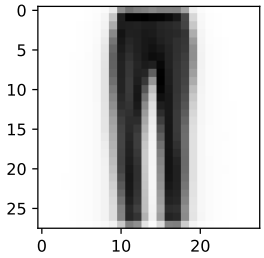


2 つ目の場合も、平均はぼやけたズボンの画像に見える。

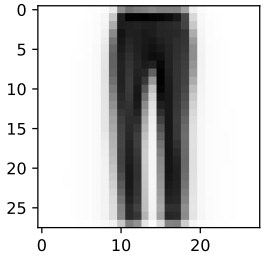
```
# Plot average trousers
d2l.plt.imshow(ave_1.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```



```
# Plot average trousers
d2l.plt.imshow(ave_1.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```




```
# Plot average trousers
d2l.plt.imshow(tf.reshape(ave_1, (28, 28)), cmap='Greys')
d2l.plt.show()
```



完全に機械学習された解法では、しきい値をデータセットから学習する。この場合は、訓練データで良さそうに見えるしきい値を自分で目分量で決めた。

```
# Print test set accuracy with eyeballed threshold
w = (ave_1 - ave_0).T
# '@' is Matrix Multiplication operator in pytorch.
predictions = X_test.reshape(2000, -1) @ (w.flatten()) > -1500000

# Accuracy
torch.mean((predictions.type(y_test.dtype) == y_test).float(), dtype=torch.
↪float64)
```

```
tensor(0.7870, dtype=torch.float64)
```

```
# Print test set accuracy with eyeballed threshold
w = (ave_1 - ave_0).T
predictions = X_test.reshape(2000, -1).dot(w.flatten()) > -1500000
```

(continues on next page)

(continued from previous page)

```
# Accuracy
np.mean(predictions.astype(y_test.dtype) == y_test, dtype=np.float64)
```

```
array(0.801, dtype=float64)
```

```
# Print test set accuracy with eyeballed threshold
w = tf.transpose(ave_1 - ave_0)
predictions = tf.reduce_sum(X_test * tf.nest.flatten(w), axis=0) > -1500000

# Accuracy
tf.reduce_mean(
    tf.cast(tf.cast(predictions, y_test.dtype) == y_test, tf.float32))
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=0.4602704>
```

25.1.4 線形変換の幾何

Section 5.3 と上の議論を通じて、ベクトル、長さ、角度の幾何についてはしっかり理解できた。しかし、まだ議論していない重要な対象が一つある。それは、行列で表される線形変換の幾何学的理解である。2 つの、場合によっては異なる高次元空間の間で、行列がデータをどう変換するのかを完全に身につけるにはかなりの練習が必要であり、この付録の範囲を超えている。しかし、2 次元から直感を積み上げ始めることはできる。

ある行列があるとする。

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}. \tag{25.1.11}$$

これを任意のベクトル $\mathbf{v} = [x, y]^\top$ に適用したいとき、掛け算を行うと

$$\begin{aligned} \mathbf{A}\mathbf{v} &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ &= \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} \\ &= x \begin{bmatrix} a \\ c \end{bmatrix} + y \begin{bmatrix} b \\ d \end{bmatrix} \\ &= x \left\{ \mathbf{A} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\} + y \left\{ \mathbf{A} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}. \end{aligned} \tag{25.1.12}$$

これは奇妙な計算に見えるかもしれない。明快だったものが、やや見通しの悪いものになったように感じられるだろう。しかし、これにより、行列が 任意のベクトルをどう変換するかを、2 つの特定のベクトル: $[1, 0]^\top$ と $[0, 1]^\top$ に対する変換の仕方で表せることがわかる。これは少し考える価値がある。本質的に

は、無限個の問題（任意の実数の組に何が起こるか）を有限個の問題（これらの特定のベクトルに何が起こるか）に還元しているのである。これらのベクトルは 基底の一例であり、空間内の任意のベクトルをこれらの 基底ベクトルの重み付き和として書ける。

特定の行列

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}. \tag{25.1.13}$$

を使うと何が起こるか描いてみよう。

特定のベクトル $\mathbf{v} = [2, -1]^\top$ を見ると、これは $2 \cdot [1, 0]^\top + -1 \cdot [0, 1]^\top$ なので、行列 A はこれを $2(\mathbf{A}[1, 0]^\top) + -1(\mathbf{A}[0, 1])^\top = 2[1, -1]^\top - [2, 3]^\top = [0, -5]^\top$ へ送ることがわかる。この論理を注意深くたどり、たとえばすべての整数対の点の格子を考えると、起こることは、行列積が格子をせん断し、回転し、拡大縮小する一方で、格子構造自体は Fig. 25.1.8 に示すように保たれる、ということである。

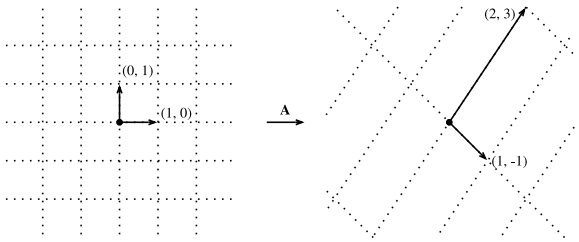


Fig. 25.1.8 与えられた基底ベクトルに作用する行列 $\mathbf{A}$。格子全体もそれに伴って移されることに注意。

これが、行列で表される線形変換について身につけるべき最も重要な直感である。行列は、空間のある部分だけを他の部分と異なるように歪めることはできない。できるのは、空間上の元の座標をせん断し、回転し、拡大縮小することだけである。

歪みが非常に大きくなることもある。たとえば、行列

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}, \tag{25.1.14}$$

は 2 次元平面全体を 1 本の直線へと圧縮する。このような変換を識別し扱うことは後の節の話題であるが、幾何学的には、これは上で見た変換の種類とは本質的に異なることがわかる。たとえば、行列 $\mathbf{A}$ の結果は元の格子へ「戻す」ことができる。しかし、行列 $\mathbf{B}$ の結果は戻せない。なぜなら、ベクトル $[1, 2]^\top$ がどこから来たのかはわからないからです—それは $[1, 1]^\top$ だったのだろうか、それとも $[0, -1]^\top$ だったのだろうか？

この図は 2×2 行列についてのものでしたが、ここで学んだことを高次元に持ち込むことを妨げるものはない。$[1, 0, \ldots, 0]$ のような同様の基底ベクトルを取り、行列がそれらをどこへ送るかを見れば、扱っている次元が何であれ、

行列積が空間全体をどう歪めるかについて感覚をつかみ始めることができる。

25.1.5 線形従属

再び行列

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}. \tag{25.1.15}$$

を考える。

これは平面全体を直線 $y = 2x$ 上に押しつぶする。ここで疑問が生じる。行列そのものを見るだけで、これを検出する方法はあるのだろうか？答えは、実際にある、である。$\mathbf{B}$ の 2 つの列を $\mathbf{b}_1 = [2, 4]^\top$ と $\mathbf{b}_2 = [-1, -2]^\top$ とする。行列 $\mathbf{B}$ によって変換されるものはすべて、行列の列の重み付き和たとえば $a_1\mathbf{b}_1 + a_2\mathbf{b}_2$ として書けることを思い出する。これを 線形結合と呼ぶ。$\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ であるという事実は、たとえば $\mathbf{b}_2$ だけを使ってこれら 2 つの列の任意の線形結合を書けることを意味する。なぜなら

$$a_1\mathbf{b}_1 + a_2\mathbf{b}_2 = -2a_1\mathbf{b}_2 + a_2\mathbf{b}_2 = (a_2 - 2a_1)\mathbf{b}_2. \tag{25.1.16}$$

だからである。

これは、ある意味で一方の列が冗長であることを意味する。空間内で一意な方向を定めていないからである。これはそれほど驚くことではない。すでにこの行列が平面全体を 1 本の直線へ押しつぶすことを見たからである。さらに、線形従属 $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ がこれを捉えていることもわかる。2 つのベクトルの対称性を高めるため、これを

$$\mathbf{b}_1 + 2 \cdot \mathbf{b}_2 = 0. \tag{25.1.17}$$

と書く。

一般に、ベクトルの集まり $\mathbf{v}_1, \ldots, \mathbf{v}_k$ が 線形従属であるとは、係数 $a_1, \ldots, a_k$ がすべて *0* ではないものが存在して

$$\sum_{i=1}^{k} a_i\mathbf{v_i} = 0. \tag{25.1.18}$$

となることをいいる。

この場合、あるベクトルを他のベクトルの組み合わせで解けるので、実質的に冗長になる。したがって、行列の列に線形従属があることは、その行列が空間をより低い次元へ圧縮していることの証拠である。線形従属がなければ、ベクトルは 線形独立であるといいる。行列の列が線形独立なら、圧縮は起こらず、その操作は元に戻せる。

25.1.6 階数

一般の $n \times m$ 行列があるとき、その行列がどの次元の空間へ写すのかを問うのは自然である。階数として知られる概念がその答えになる。前節では、線形従属が空間の低次元への圧縮の証拠であることを述べたので、これを階数の概念の定義に使える。特に、行列 $\mathbf{A}$ の階数とは、列の部分集合の中で線形独立な列の最大数である。たとえば、行列

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}, \tag{25.1.19}$$

は 2 つの列が線形従属なので $\text{rank}(B) = 1$ である。しかし、どちらか一方の列だけなら線形従属ではない。より難しい例として、

$$\mathbf{C} = \begin{bmatrix} 1 & 3 & 0 & -1 & 0 \\ -1 & 0 & 1 & 1 & -1 \\ 0 & 3 & 1 & 0 & -1 \\ 2 & 3 & -1 & -2 & 1 \end{bmatrix}, \tag{25.1.20}$$

を考えると、たとえば最初の 2 列は線形独立であるが、3 列の 4 通りの組はどれも従属であることから、$\mathbf{C}$ の階数は 2 であると示せる。

この手順は、述べたとおり非常に非効率である。与えられた行列の列のすべての部分集合を調べる必要があり、したがって列数に対して指数時間になる可能性がある。後で、行列の階数を計算するより計算効率のよい方法を見るが、今のところは、この概念がよく定義されており、その意味を理解するにはこれで十分である。

25.1.7 可逆性

上で、線形従属な列を持つ行列との掛け算は元に戻せない、すなわち入力を常に復元できる逆操作は存在しないことを見た。しかし、フルランク行列（すなわち、階数が n の $n \times n$ 行列 $\mathbf{A}$）との掛け算は、常に元に戻せるはずである。行列

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{25.1.21}$$

を考える。

これは対角成分が 1 で、それ以外が 0 の行列である。これを 単位行列と呼ぶ。適用してもデータを変えない行列である。行列 $\mathbf{A}$ が行ったことを打ち消す行列を見つけるには、次を満たす行列 $\mathbf{A}^{-1}$ を見つけたいのである。

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}. \tag{25.1.22}$$

これを連立方程式として見ると、未知数は $n \times n$ （$\mathbf{A}^{-1}$ の各要素）で、方程式も $n \times n$ （$\mathbf{A}^{-1}\mathbf{A}$ の各要素と $\mathbf{I}$ の各要素の間で成り立つべき等式）あるので、一般

には解が存在すると期待できる。実際、次の節では 行列式と呼ばれる量を見るが、その値が 0 でなければ解を見つけられるという性質がある。このような行列 $\mathbf{A}^{-1}$ を 逆行列と呼ぶ。例として、$\mathbf{A}$ が一般の 2×2 行列

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{25.1.23}$$

なら、その逆行列は

$$\frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{25.1.24}$$

であることがわかる。

これが本当にそうかは、上の式で与えられる逆行列を掛けると実際にうまくいくことを確認すればよいである。

```
M = torch.tensor([[1, 2], [1, 4]], dtype=torch.float32)
M_inv = torch.tensor([[2, -1], [-0.5, 0.5]])
M_inv @ M
```

```
tensor([[1., 0.],
        [0., 1.]])
```

```
M = np.array([[1, 2], [1, 4]])
M_inv = np.array([[2, -1], [-0.5, 0.5]])
M_inv.dot(M)
```

```
array([[1., 0.],
       [0., 1.]])
```

```
M = tf.constant([[1, 2], [1, 4]], dtype=tf.float32)
M_inv = tf.constant([[2, -1], [-0.5, 0.5]])
tf.matmul(M_inv, M)
```

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[1., 0.],
       [0., 1.]], dtype=float32)>
```

数値的な問題

行列の逆行列は理論上は有用であるが、実際には問題を解くのに行列の逆行列を 使うことは、たいてい望ましくないと言わなければならない。一般に、次のような線形方程式を解くには、

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{25.1.25}$$

逆行列を計算して掛ける

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \tag{25.1.26}$$

よりも、はるかに数値的に安定なアルゴリズムがある。

小さな数で割ると数値的不安定性が生じうるのと同様に、低階数に近い行列の逆行列を求めることも不安定になりえる。

さらに、行列 $\mathbf{A}$ が 疎、つまり非ゼロ値を少数しか含まないことはよくある。例を調べれば、逆行列が疎であるとは限らないことがわかるだろう。たとえ $\mathbf{A}$ が 100 万 ×100 万の行列で、非ゼロ要素が 500 万個しかない（したがって保存する必要があるのはその 500 万個だけ）としても、逆行列は通常ほとんどすべての要素が非ゼロになり、1M^2 個、つまり 1 兆個の要素すべてを保存する必要が出てきます！

線形代数を扱うときにしばしば遭遇する厄介な数値問題をここで完全に掘り下げる時間はないが、どのような場合に注意して進むべきかについての直感を与えたいと思いる。一般には、実務で逆行列を避けるのがよい経験則である。

25.1.8 行列式

線形代数の幾何学的な見方は、行列式として知られる基本量を直感的に解釈する方法を与えてくれる。先ほどの格子の図を、今度は強調された領域つきで考えます（Fig. 25.1.9）。

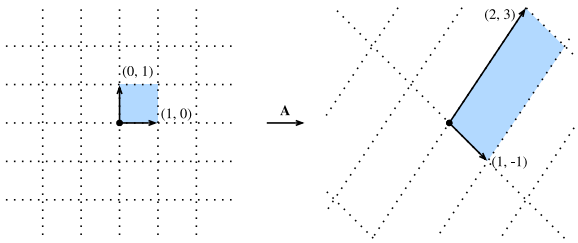


Fig. 25.1.9 再び格子を歪める行列 $\mathbf{A}$。今度は、強調された正方形に何が起こるかに特に注目したい。

強調された正方形を見よ。これは辺が $(0,1)$ と $(1,0)$ で与えられる正方形であり、面積は 1 である。$\mathbf{A}$ がこの正方形を変換した後、それは平行四辺形になる。この平行四辺形が元の面積と同じである理由はなく、実際、ここで示した特定の場合

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}, \tag{25.1.27}$$

では、この平行四辺形の面積を求めると 5 になることは、座標幾何の練習問題である。

一般に、行列

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{25.1.28}$$

があるとき、結果として得られる平行四辺形の面積は $ad - bc$ であることが計算によりわかる。この面積を 行列式と呼ぶ。

簡単な例で確認してみよう。

```
torch.det(torch.tensor([[1, -1], [2, 3]], dtype=torch.float32))
```

```
tensor(5.)
```

```
import numpy as np
np.linalg.det(np.array([[1, -1], [2, 3]]))
```

```
5.000000000000001
```

```
tf.linalg.det(tf.constant([[1, -1], [2, 3]], dtype=tf.float32))
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=5.0>
```

鋭い読者は、この式が 0 にも、負にもなりうることに気づくだろう。負の項については、これは一般に数学で採用される慣習の問題である。行列が図形を反転させるなら、面積は符号反転すると考える。では、行列式が 0 のときに何がわかるか見てみよう。

次を考える。

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}. \tag{25.1.29}$$

この行列の行列式を計算すると、$2 \cdot (-2) - 4 \cdot (-1) = 0$ である。上での理解に照らすと、これは納得できる。$\mathbf{B}$ は元の図の正方形を面積 0 の線分へと圧縮する。そして実際、変換後に面積が 0 になるのは、低次元空間へ圧縮される場合に限られる。したがって、次の結果が成り立ちる。行列 A が可逆であるのは、行列式が 0 でないとき、かつそのときに限りる。

最後に、平面上に何らかの図形が描かれていると想像する。コンピュータ科学者のように考えると、その図形を小さな正方形の集合に分解できるので、図形の面積は本質的には分解に含まれる正方形の数にすぎない。この図形を行列で変換すると、各正方形は平行四辺形へ送られ、それぞれの面積は行列式で与えられる。したがって、任意の図形について、行列式は行列がその図形の面積をどれだけ拡大縮小するかを表す（符号付きの）数を与えることがわかる。

より大きな行列の行列式を計算するのは骨が折れるが、直感は同じである。行列式は、$n \times n$ 行列が n 次元体積をどれだけ拡大縮小するかを与える係数であり続ける。

25.1.9 テンソルと一般的な線形代数操作

Section 5.3 ではテンソルの概念を導入した。この節では、テンソル縮約（行列積に相当するテンソル版）をさらに深く掘り下げ、それが多くの行列・ベクトル操作を統一的に捉える視点を与えることを見る。

行列やベクトルでは、それらを掛けてデータを変換する方法を知っていた。テンソルが私たちにとって有用であるためには、同様の定義が必要である。行列積を考えてみよう。

$$\mathbf{C} = \mathbf{AB}, \tag{25.1.30}$$

あるいは同値に

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}. \tag{25.1.31}$$

このパターンはテンソルにも繰り返し使える。テンソルでは、何について和を取るかを一意に普遍的に選べるわけではないので、どの添字について和を取るのかを正確に指定する必要がある。たとえば、

$$y_{il} = \sum_{jk} x_{ijkl} a_{jk}. \tag{25.1.32}$$

のように考えられる。

このような変換を テンソル縮約と呼ぶ。これは、行列積だけよりもはるかに柔軟な変換の族を表せる。

よく使われる記法上の簡略化として、和を取る添字は式中に 2 回以上現れる添字ちょうどであることに注目できる。そのため、人々はしばしば アインシュタイン記法を用い、繰り返し現れる添字に対して和が暗黙に取られるようにする。これにより、次の簡潔な表現が得られる。

$$y_{il} = x_{ijkl} a_{jk}. \tag{25.1.33}$$

線形代数の一般的な例

これまで見てきた線形代数の定義の多くが、この圧縮されたテンソル記法でどう表せるか見てみよう。

- $\mathbf{v} \cdot \mathbf{w} = \sum_i v_i w_i$
- $\|\mathbf{v}\|_2^2 = \sum_i v_i v_i$
- $(\mathbf{Av})_i = \sum_j a_{ij} v_j$
- $(\mathbf{AB})_{ik} = \sum_j a_{ij} b_{jk}$

- $\text{tr}(\mathbf{A}) = \sum_i a_{ii}$

このようにして、無数の特殊な記法を短いテンソル表現に置き換えられる。

コードで表す

テンソルはコード中でも柔軟に操作できる。Section 5.3 で見たように、以下のようにテンソルを作成できる。

```
# Define tensors
B = torch.tensor([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
A = torch.tensor([[1, 2], [3, 4]])
v = torch.tensor([1, 2])

# Print out the shapes
A.shape, B.shape, v.shape
```

```
(torch.Size([2, 2]), torch.Size([2, 2, 3]), torch.Size([2]))
```

```
# Define tensors
B = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
A = np.array([[1, 2], [3, 4]])
v = np.array([1, 2])

# Print out the shapes
A.shape, B.shape, v.shape
```

```
((2, 2), (2, 2, 3), (2,))
```

```
# Define tensors
B = tf.constant([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
A = tf.constant([[1, 2], [3, 4]])
v = tf.constant([1, 2])

# Print out the shapes
A.shape, B.shape, v.shape
```

```
(TensorShape([2, 2]), TensorShape([2, 2, 3]), TensorShape([2]))
```

アインシュタインの総和は直接実装されている。アインシュタイン総和に現れる添字は文字列として渡し、その後に作用させるテンソルを続ける。たとえば、行列積を実装するには、上で見たアインシュタイン総和（$\mathbf{Av} = a_{ij}v_j$）を考え、添字そのものを取り除くと次の実装になる。

```
# Reimplement matrix multiplication
torch.einsum("ij, j -> i", A, v), A@v
```

```
(tensor([ 5, 11]), tensor([ 5, 11]))
```

```
# Reimplement matrix multiplication
np.einsum("ij, j -> i", A, v), A.dot(v)
```

```
(array([ 5, 11]), array([ 5, 11]))
```

```
# Reimplement matrix multiplication
tf.einsum("ij, j -> i", A, v), tf.matmul(A, tf.reshape(v, (2, 1)))
```

```
(<tf.Tensor: shape=(2,), dtype=int32, numpy=array([ 5, 11], dtype=int32)>,
 <tf.Tensor: shape=(2, 1), dtype=int32, numpy=
 array([[ 5],
        [11]], dtype=int32)>)
```

これは非常に柔軟な記法である。たとえば、伝統的には次のように書くものを計算したいとする。

$$c_{kl} = \sum_{ij} \mathbf{b}_{ijk} \mathbf{a}_{il} v_j. \tag{25.1.34}$$

これはアインシュタイン総和を使って次のように実装できる。

```
torch.einsum("ijk, il, j -> kl", B, A, v)
```

```
tensor([[ 90, 126],
        [102, 144],
        [114, 162]])
```

```
np.einsum("ijk, il, j -> kl", B, A, v)
```

```
array([[ 90, 126],
       [102, 144],
       [114, 162]])
```

```
tf.einsum("ijk, il, j -> kl", B, A, v)
```

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[ 90, 126],
       [102, 144],
       [114, 162]], dtype=int32)>
```

この記法は人間にとって読みやすく効率的であるが、何らかの理由でテンソル

縮約をプログラム的に生成する必要がある場合には、ややかさばりる。このため、einsum には各テンソルに整数の添字を与える別の記法もある。たとえば、同じテンソル縮約は次のようにも書ける。

```
# PyTorch does not support this type of notation.
```

```
np.einsum(B, [0, 1, 2], A, [0, 3], v, [1], [2, 3])
```

```
array([[ 90, 126],
       [102, 144],
       [114, 162]])
```

```
# TensorFlow does not support this type of notation.
```

どちらの記法でも、コード中でテンソル縮約を簡潔かつ効率的に表現できる。

25.1.10 まとめ

- ベクトルは、空間内の点または方向として幾何学的に解釈できる。
- 内積は、任意に高次元の空間における角度の概念を定義する。
- 超平面は、直線や平面の高次元一般化である。分類タスクの最後の段階でよく使われる決定平面を定義するのに使える。
- 行列積は、基礎となる座標を一様に歪めるものとして幾何学的に解釈できる。これはベクトルを変換する非常に制限された、しかし数学的には明快な方法である。
- 線形従属は、ベクトルの集まりが期待より低い次元空間にあることを見分ける方法である（たとえば、2 次元空間に 3 つのベクトルがある場合）。行列の階数は、その列のうち線形独立な最大部分集合の大きさである。
- 行列の逆行列が定義されるとき、行列の反転により最初の行列の作用を打ち消す別の行列を見つけられる。逆行列は理論上有用だが、数値的不安定性のため実務では注意が必要である。
- 行列式により、行列が空間をどれだけ拡大または縮小するかを測れる。行列式が 0 でないことは可逆（非特異）行列であることを意味し、行列式が 0 であることは非可逆（特異）行列であることを意味する。
- テンソル縮約とアインシュタイン総和は、機械学習で見られる多くの計算を表すための、簡潔で明快な記法を与える。

25.1.11 演習

1. 次のベクトルのなす角は何か。

$$\vec{v}_1 = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 2 \end{bmatrix}, \qquad \vec{v}_2 = \begin{bmatrix} 3 \\ 1 \\ 0 \\ 1 \end{bmatrix}? \tag{25.1.35}$$

2. 真か偽か：$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ と $\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$ は互いに逆行列であるか？

3. 平面上に面積 100m^2 の図形を描いたとする。次の行列で図形を変換した後の面積はどうなるか。

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}. \tag{25.1.36}$$

4. 次のベクトル集合のうち、線形独立なのはどれか。

- $\left\{\begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix}\right\}$
- $\left\{\begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}\right\}$
- $\left\{\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}\right\}$

5. ある値 a, b, c, d の選び方に対して、$A = \begin{bmatrix} c \\ d \end{bmatrix} \cdot \begin{bmatrix} a & b \end{bmatrix}$ と書ける行列があるとする。真か偽か：このような行列の行列式は常に 0 である。

6. ベクトル $e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ と $e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ は直交している。Ae_1 と Ae_2 が直交するための行列 A の条件は何か？

7. 任意の行列 A について、$\text{tr}(\mathbf{A}^4)$ をアインシュタイン記法でどのように書けるか？

25.2 固有分解

固有値は、線形代数を学ぶ際にしばしば最も有用な概念の一つである。しかし、初心者にとってはその重要性を見落としやすいものでもある。以下では、

固有分解を導入し、それがなぜこれほど重要なのかを少しでも伝えたいと思いる。

次の成分をもつ行列 A があるとする。

$$\mathbf{A} = \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix}. \tag{25.2.1}$$

A を任意のベクトル $\mathbf{v} = [x, y]^\top$ に適用すると、$\mathbf{Av} = [2x, -y]^\top$ というベクトルが得られる。これは直感的には、ベクトルを x 方向に 2 倍に引き伸ばし、その後 y 方向に反転させる、と解釈できる。

しかし、ある種のベクトルでは、何かが変わらないまま残る。すなわち、$[1, 0]^\top$ は $[2, 0]^\top$ に写され、$[0, 1]^\top$ は $[0, -1]^\top$ に写される。これらのベクトルは依然として同じ直線上にあり、変化するのは行列がそれらをそれぞれ 2 と -1 の倍率で伸縮することだけである。このようなベクトルを固有ベクトル、その伸縮の倍率を固有値と呼ぶ。

一般に、ある数 λ とベクトル $\mathbf{v}$ が存在して

$$\mathbf{Av} = \lambda\mathbf{v}. \tag{25.2.2}$$

となるとき、$\mathbf{v}$ は A の固有ベクトルであり、λ は固有値であると言う。

25.2.1 固有値の求め方

それらをどのように求めるかを考えてみよう。両辺から $\lambda\mathbf{v}$ を引き、さらにベクトルをくくり出すと、上式は次と同値であることがわかる。

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0. \tag{25.2.3}$$

(25.2.3) が成り立つためには、$(\mathbf{A} - \lambda\mathbf{I})$ がある方向をゼロまで押しつぶす必要がある。したがってこれは可逆ではなく、ゆえに行列式はゼロである。したがって、$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ となる λ を求めることで、固有値を見つけることができる。固有値が求まれば、$\mathbf{Av} = \lambda\mathbf{v}$ を解くことで、対応する固有ベクトルを求められる。

例

もう少し難しい行列で見てみよう。

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix}. \tag{25.2.4}$$

$\det(\mathbf{A}-\lambda\mathbf{I}) = 0$ を考えると、これは多項式方程式 $0 = (2-\lambda)(3-\lambda)-2 = (4-\lambda)(1-\lambda)$ と同値であることがわかる。したがって、固有値は 4 と 1 の 2 つである。対応するベクトルを求めるには、次を解く必要がある。

$$\begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \text{ and } \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4x \\ 4y \end{bmatrix}. \tag{25.2.5}$$

それぞれ、ベクトル $[1, -1]^\top$ と $[1, 2]^\top$ で解ける。

これを組み込みの numpy.linalg.eig ルーチンを使ってコードで確認できる。

```
%matplotlib inline
from d2l import torch as d2l
from IPython import display
import torch

torch.linalg.eig(torch.tensor([[2, 1], [2, 3]], dtype=torch.float64))
```

```
torch.return_types.linalg_eig(
eigenvalues=tensor([1.+0.j, 4.+0.j], dtype=torch.complex128),
eigenvectors=tensor([[-0.7071+0.j, -0.4472+0.j],
        [ 0.7071+0.j, -0.8944+0.j]], dtype=torch.complex128))
```

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
import numpy as np

np.linalg.eig(np.array([[2, 1], [2, 3]]))
```

```
(array([1., 4.]),
 array([[-0.70710678, -0.4472136 ],
        [ 0.70710678, -0.89442719]]))
```

```
%matplotlib inline
from d2l import tensorflow as d2l
from IPython import display
import tensorflow as tf

tf.linalg.eig(tf.constant([[2, 1], [2, 3]], dtype=tf.float64))
```

```
(<tf.Tensor: shape=(2,), dtype=complex128, numpy=array([1.+0.j, 4.+0.j])>,
 <tf.Tensor: shape=(2, 2), dtype=complex128, numpy=
 array([[-0.70710678+0.j, -0.4472136 +0.j],
        [ 0.70710678+0.j, -0.89442719+0.j]])>)
```

numpy は固有ベクトルを長さ 1 に正規化する一方で、ここでは任意の長さを取っていた。さらに、符号の選び方も任意である。しかし、計算されたベクトルは、同じ固有値に対して手計算で求めたものと平行である。

25.2.2 行列の分解

前の例をもう一歩進めてみよう。次を

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix}, \tag{25.2.6}$$

行列 $\mathbf{A}$ の固有ベクトルを列にもつ行列とする。さらに

$$\mathbf{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}, \tag{25.2.7}$$

を対応する固有値を対角に並べた行列とする。すると、固有値と固有ベクトルの定義から

$$\mathbf{AW} = \mathbf{W\Sigma}. \tag{25.2.8}$$

が成り立ちる。

行列 W は可逆なので、両辺の右から W^{-1} を掛けると、次のように書ける。

$$\mathbf{A} = \mathbf{W\Sigma W}^{-1}. \tag{25.2.9}$$

次節でこれのいくつかの良い帰結を見るが、今のところは、このような分解は線形独立な固有ベクトルを十分な数だけ見つけられる限り（したがって W が可逆である限り）存在する、ということだけ知っておけば十分である。

25.2.3 固有分解に対する演算

(25.2.9) の固有分解の良い点の一つは、通常出会う多くの演算を固有分解の形できれいに書けることである。最初の例として、次を考える。

$$\mathbf{A}^n = \overbrace{\mathbf{A}\cdots\mathbf{A}}^{n \text{ times}} = \overbrace{(\mathbf{W\Sigma W}^{-1})\cdots(\mathbf{W\Sigma W}^{-1})}^{n \text{ times}} = \mathbf{W}\overbrace{\mathbf{\Sigma}\cdots\mathbf{\Sigma}}^{n \text{ times}}\mathbf{W}^{-1} = \mathbf{W\Sigma}^n\mathbf{W}^{-1}. \tag{25.2.10}$$

これは、行列の任意の正のべきについて、固有分解では固有値を同じべきに上げるだけでよいことを示している。負のべきについても同様に示せるので、行列を逆にしたいなら、各固有値を逆数にするだけでよいことになる。

$$\mathbf{A}^{-1} = \mathbf{W\Sigma}^{-1}\mathbf{W}^{-1}, \tag{25.2.11}$$

つまり、各固有値を逆にすればよいのである。これは各固有値がゼロでない限り成り立つので、可逆であることとゼロ固有値を持たないことは同じだとわかる。

実際、さらに調べると、$\lambda_1, \ldots, \lambda_n$ がある行列の固有値であるとき、その行列の行列式は

$$\det(\mathbf{A}) = \lambda_1 \cdots \lambda_n, \tag{25.2.12}$$

すなわち全固有値の積になる。これは直感的にも納得できる。$\mathbf{W}$ がどのような伸縮をしても、W^{-1} がそれを打ち消すので、最終的に起こる伸縮は対角行列 $\mathbf{\Sigma}$ による掛け算だけである。そして対角行列は、対角成分の積だけ体積を伸縮させる。

最後に、階数は行列の線形独立な列の最大数だったことを思い出する。固有分解を詳しく見ると、階数は $\mathbf{A}$ のゼロでない固有値の数と同じであることがわかる。

例はいくらでも続けられるが、要点はおそらく明らかだろう。固有分解は多くの線形代数計算を簡単にし、多くの数値アルゴリズムや線形代数で行う解析の基礎となる基本的な操作なのである。

25.2.4 対称行列の固有分解

上の手順がうまく働くほど十分な数の線形独立な固有ベクトルを見つけられない場合もある。たとえば次の行列

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \tag{25.2.13}$$

は、固有ベクトルをただ 1 つ、すなわち $(1, 0)^\top$ しか持たない。このような行列を扱うには、ここで扱える範囲を超えるより高度な手法（たとえばジョルダン標準形や特異値分解）が必要である。そのため、しばしば固有ベクトルの完全な集合の存在を保証できる行列に話を限定する必要がある。

最もよく現れるのは対称行列の族で、これは $\mathbf{A} = \mathbf{A}^\top$ を満たす行列である。この場合、W を直交行列、すなわち列ベクトルがすべて長さ 1 で互いに直交し、$\mathbf{W}^\top = \mathbf{W}^{-1}$ を満たす行列として取ることができ、固有値はすべて実数になる。したがって、この特別な場合には (25.2.9) を

$$\mathbf{A} = \mathbf{W}\mathbf{\Sigma}\mathbf{W}^\top. \tag{25.2.14}$$

と書ける。

25.2.5 ガーシュゴリンの円定理

固有値は直感的に理解するのが難しいことがよくある。任意の行列を与えられても、計算せずに固有値について言えることはあまりない。しかし、対角成分が大きい場合に、よい近似を簡単に与えてくれる定理が一つある。

$\mathbf{A} = (a_{ij})$ を任意の正方行列（$n \times n$）とする。$r_i = \sum_{j \neq i} |a_{ij}|$ と定義する。$\mathcal{D}_i$ を、中心が a_{ii}、半径が r_i の複素平面上の円板とする。すると、$\mathbf{A}$ のすべての固有値はどれか一つの $\mathcal{D}_i$ に含まれる。

これは少し分かりにくいので、例を見てみよう。次の行列を考える。

$$\mathbf{A} = \begin{bmatrix} 1.0 & 0.1 & 0.1 & 0.1 \\ 0.1 & 3.0 & 0.2 & 0.3 \\ 0.1 & 0.2 & 5.0 & 0.5 \\ 0.1 & 0.3 & 0.5 & 9.0 \end{bmatrix}. \tag{25.2.15}$$

$r_1 = 0.3, r_2 = 0.6, r_3 = 0.8, r_4 = 0.9$ である。この行列は対称なので、固有値はすべて実数である。したがって、固有値は次の範囲のいずれかに入る。

$$[a_{11} - r_1, a_{11} + r_1] = [0.7, 1.3], \tag{25.2.16}$$

$$[a_{22} - r_2, a_{22} + r_2] = [2.4, 3.6], \tag{25.2.17}$$

$$[a_{33} - r_3, a_{33} + r_3] = [4.2, 5.8], \tag{25.2.18}$$

$$[a_{44} - r_4, a_{44} + r_4] = [8.1, 9.9]. \tag{25.2.19}$$

数値計算を行うと、固有値はおよそ 0.99, 2.97, 4.95, 9.08 であり、与えられた範囲の中に十分収まっていることがわかる。

```
A = torch.tensor([[1.0, 0.1, 0.1, 0.1],
                  [0.1, 3.0, 0.2, 0.3],
                  [0.1, 0.2, 5.0, 0.5],
                  [0.1, 0.3, 0.5, 9.0]])

v, _ = torch.linalg.eig(A)
v
```

```
tensor([0.9923+0.j, 9.0803+0.j, 4.9539+0.j, 2.9734+0.j])
```

```
A = np.array([[1.0, 0.1, 0.1, 0.1],
              [0.1, 3.0, 0.2, 0.3],
              [0.1, 0.2, 5.0, 0.5],
              [0.1, 0.3, 0.5, 9.0]])

v, _ = np.linalg.eig(A)
v
```

```
array([9.08033648, 0.99228545, 4.95394089, 2.97343718])
```

```
A = tf.constant([[1.0, 0.1, 0.1, 0.1],
                 [0.1, 3.0, 0.2, 0.3],
                 [0.1, 0.2, 5.0, 0.5],
                 [0.1, 0.3, 0.5, 9.0]])

v, _ = tf.linalg.eigh(A)
v
```

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([0.99228525, 2.9734395 , 4.
↪953943  , 9.080336  ], dtype=float32)>
```

このようにして、固有値は近似できる。そして、対角成分が他のすべての要素よりかなり大きい場合には、その近似はかなり正確になる。

小さなことではあるが、固有分解のような複雑で繊細な話題では、少しでも直感的な理解を得られるのは良いことである。

25.2.6 有用な応用: 反復写像の成長

固有ベクトルが原理的に何であるかを理解したので、ニューラルネットワークの振る舞いの中心的な問題である適切な重み初期化に対して、どのように深い理解を与えられるかを見てみよう。

長期的な振る舞いとしての固有ベクトル

深層ニューラルネットワークの初期化に関する完全な数学的検討は本書の範囲を超えるが、ここではそのおもちゃ版を見て、固有値がこれらのモデルの働きをどう理解する助けになるかを見てみよう。ご存じのように、ニューラルネットワークは線形変換の層と非線形演算を交互に挟むことで動作する。ここでは簡単のため、非線形性はなく、変換は単一の行列演算 A の反復だと仮定する。すると、モデルの出力は

$$\mathbf{v}_{out} = \mathbf{A} \cdot \mathbf{A} \cdots \mathbf{A}\mathbf{v}_{in} = \mathbf{A}^N \mathbf{v}_{in}. \tag{25.2.20}$$

これらのモデルを初期化するとき、A はガウス分布に従う成分をもつランダム行列として取られるので、そのような行列を一つ作ってみよう。具体的には、平均 0、分散 1 のガウス分布に従う 5×5 行列から始める。

```
torch.manual_seed(42)

k = 5
A = torch.randn(k, k, dtype=torch.float64)
A
```

```
tensor([[ 0.2996,  0.2424,  0.2832, -0.2329,  0.6712],
        [ 0.7818, -1.7903, -1.7484,  0.1735, -0.1182],
        [-1.7446, -0.4695,  0.4573,  0.5177, -0.2771],
        [-0.6641,  0.6551,  0.2616, -1.5265, -0.3311],
        [-0.6378,  0.1072,  0.7096,  0.3009, -0.2869]], dtype=torch.float64)
```

```
np.random.seed(8675309)

k = 5
A = np.random.randn(k, k)
A
```

```
array([[ 0.58902366,  0.73311856, -1.1621888 , -0.55681601, -0.77248843],
       [-0.16822143, -0.41650391, -1.37843129,  0.74925588,  0.17888446],
       [ 0.69401121, -1.9780535 , -0.83381434,  0.56437344,  0.31201299],
       [-0.87334496,  0.15601291, -0.38710108, -0.23920821,  0.88850104],
       [ 1.29385371, -0.76774106,  0.20131613,  0.91800842,  0.38974115]])
```

```
k = 5
A = tf.random.normal((k, k), dtype=tf.float64)
A
```

```
<tf.Tensor: shape=(5, 5), dtype=float64, numpy=
array([[ 0.72841264, -0.07946832, -0.55780591,  0.25373236,  1.79470718],
       [ 0.59646403, -0.01602192,  0.24092203, -0.0951461 ,  0.04742508],
       [ 1.09927512,  0.04417023,  1.5446829 , -0.84427998,  1.04076418],
       [ 2.67344098,  0.61957738,  1.56904466, -0.55098639,  0.70267397],
       [-0.05988268,  1.06744539,  0.70405254, -2.25065308, -1.73487384]])>
```

ランダムデータに対する振る舞い

おもちゃモデルでは簡単のため、入力するデータベクトル $\mathbf{v}_{in}$ はランダムな 5 次元ガウスベクトルだと仮定する。何が起こってほしいかを考えてみよう。文脈として、画像のような入力データをその画像が猫の写真である確率のような予測に変換しようとする一般的な機械学習問題を考えてみる。もし $\mathbf{A}$ を繰り返し適用するとランダムベクトルが非常に長く伸びるなら、入力の小さな変化が出力の大きな変化へと増幅される。つまり、入力画像のごく小さな修正がまったく異なる予測につながってしまうのである。これは正しくなさそうです！

逆に、$\mathbf{A}$ がランダムベクトルを短く縮めてしまうなら、多くの層を通した後でベクトルは実質的に消えてしまい、出力は入力に依存しなくなる。これも明らかに正しくない！

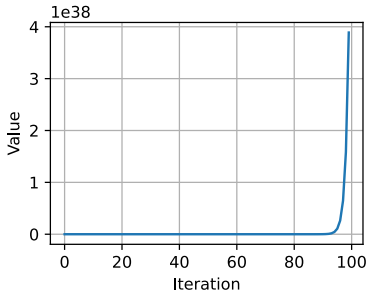
出力が入力に応じて変化するが、変化しすぎないようにするには、成長と減衰の狭い境界を歩かなければならない。

ランダムな入力ベクトルに対して行列 $\mathbf{A}$ を繰り返し掛け、ノルムを追跡すると何が起こるか見てみよう。

```
# Calculate the sequence of norms after repeatedly applying `A`
v_in = torch.randn(k, 1, dtype=torch.float64)

norm_list = [torch.norm(v_in).item()]
for i in range(1, 100):
    v_in = A @ v_in
    norm_list.append(torch.norm(v_in).item())

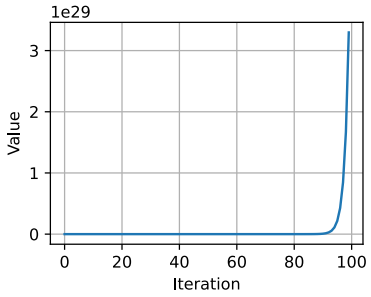
d2l.plot(torch.arange(0, 100), norm_list, 'Iteration', 'Value')
```



```
# Calculate the sequence of norms after repeatedly applying `A`
v_in = np.random.randn(k, 1)

norm_list = [np.linalg.norm(v_in)]
for i in range(1, 100):
    v_in = A.dot(v_in)
    norm_list.append(np.linalg.norm(v_in))

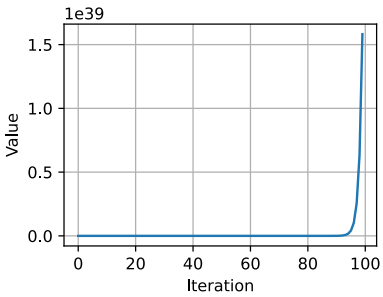
d2l.plot(np.arange(0, 100), norm_list, 'Iteration', 'Value')
```




```
# Calculate the sequence of norms after repeatedly applying `A`
v_in = tf.random.normal((k, 1), dtype=tf.float64)

norm_list = [tf.norm(v_in).numpy()]
for i in range(1, 100):
    v_in = tf.matmul(A, v_in)
    norm_list.append(tf.norm(v_in).numpy())

d2l.plot(tf.range(0, 100), norm_list, 'Iteration', 'Value')
```



ノルムが制御不能に増大している！実際、比の列を取ると、あるパターンが見えてきる。

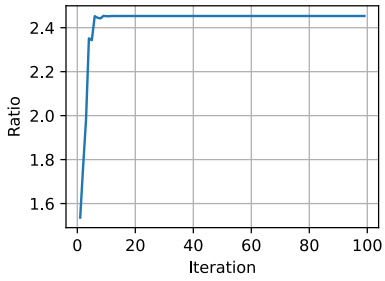
```
# Compute the scaling factor of the norms
norm_ratio_list = []
for i in range(1, 100):
```

(continues on next page)

(continued from previous page)

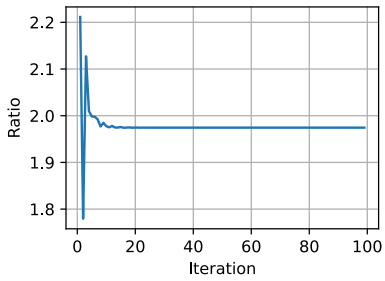
```
    norm_ratio_list.append(norm_list[i]/norm_list[i - 1])

d2l.plot(torch.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```




```
# Compute the scaling factor of the norms
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i - 1])

d2l.plot(np.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```



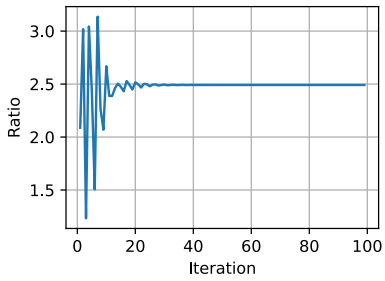

```
# Compute the scaling factor of the norms
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i - 1])

d2l.plot(tf.range(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```

上の計算の最後の部分を見ると、ランダムベクトルは `1.974459321485[...]` 倍に伸びていることがわかる。末尾の部分は少し変動するが、伸長率は安定している。

固有ベクトルとの関係

固有ベクトルと固有値は、何かがどれだけ伸びるかに対応することを見てきたが、それは特定のベクトルと特定の伸長についてでした。では、$\mathbf{A}$ に対してそ



れらが何であるかを見てみよう。ここで少し注意がある。すべてを見るには、複素数まで行く必要があることがわかる。複素数は伸長と回転だと考えられる。複素数のノルム（実部と虚部の二乗和の平方根）を取ることで、その伸長率を測れる。ついでに並べ替えよう。

```
# Compute the eigenvalues
eigs = torch.linalg.eig(A).eigenvalues.tolist()
norm_eigs = [torch.abs(torch.tensor(x)) for x in eigs]
norm_eigs.sort()
print(f'norms of eigenvalues: {norm_eigs}')
```

```
norms of eigenvalues: [tensor(0.3490), tensor(1.1296), tensor(1.1296),␣
↪tensor(1.1828), tensor(2.4532)]
```

```
# Compute the eigenvalues
eigs = np.linalg.eigvals(A).tolist()
norm_eigs = [np.absolute(x) for x in eigs]
norm_eigs.sort()
print(f'norms of eigenvalues: {norm_eigs}')
```

```
norms of eigenvalues: [0.8786205280381857, 1.2757952665062624, 1.
↪4983381517710659, 1.4983381517710659, 1.974459321485074]
```

```
# Compute the eigenvalues
eigs = tf.linalg.eigh(A)[0].numpy().tolist()
norm_eigs = [tf.abs(tf.constant(x, dtype=tf.float64)) for x in eigs]
norm_eigs.sort()
print(f'norms of eigenvalues: {norm_eigs}')
```

```
norms of eigenvalues: [<tf.Tensor: shape=(), dtype=float64, numpy=0.
↪4777341517950846>, <tf.Tensor: shape=(), dtype=float64, numpy=1.
↪0372038405780875>, <tf.Tensor: shape=(), dtype=float64, numpy=1.
↪3857880162894853>, <tf.Tensor: shape=(), dtype=float64, numpy=4.
↪366256554335739>, <tf.Tensor: shape=(), dtype=float64, numpy=4.
↪524193144166288>]
```

観察

ここで少し意外なことが起きているのがわかる。ランダムベクトルに対して行列 **A** を長期的に適用したときの伸長として先ほど特定したその数は、まさに（小数点以下 13 桁まで正確に！）**A** の最大固有値なのである。これは明らかに偶然ではない！

しかし、これを幾何学的に考えると、納得がいきる。ランダムベクトルを考えよう。このランダムベクトルはあらゆる方向を少しずつ向いているので、特に、**A** の最大固有値に対応する固有ベクトルと少なくとも少しは同じ方向を向いている。これは非常に重要なので、主固有値および主固有ベクトルと呼ばれる。**A** を適用すると、ランダムベクトルは考えられるすべての方向に伸びるが、最も強く伸びるのはこの主固有ベクトルに対応する方向である。つまり、A を適用した後では、ランダムベクトルはより長くなり、主固有ベクトルにより近い方向を向くようになる。行列を何度も適用すると、主固有ベクトルとの整列はどんどん近づき、実用上は、ランダムベクトルは主固有ベクトルへと変換されたとみなせるようになる！実際、このアルゴリズムは、行列の最大固有値と固有ベクトルを求める べき乗法の基礎になっている。詳細は、たとえば (Golub and Van Loan, 1996) を参照する。

正規化の修正

さて、上の議論から、ランダムベクトルがまったく伸びたり縮んだりしないことが望ましい、つまりランダムベクトルが処理全体を通しておおむね同じ大きさを保つことが望ましい、と結論した。そのために、行列をこの主固有値で再スケーリングし、最大固有値が 1 になるようにする。この場合に何が起こるか見てみよう。

```
# Rescale the matrix `A`
A /= norm_eigs[-1]

# Do the same experiment again
v_in = torch.randn(k, 1, dtype=torch.float64)

norm_list = [torch.norm(v_in).item()]
for i in range(1, 100):
    v_in = A @ v_in
    norm_list.append(torch.norm(v_in).item())

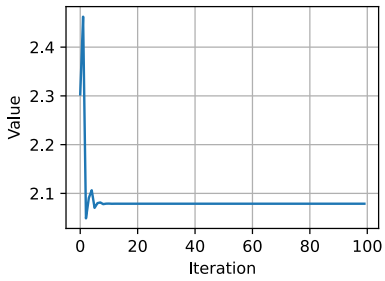
d2l.plot(torch.arange(0, 100), norm_list, 'Iteration', 'Value')
```

```
# Rescale the matrix `A`
A /= norm_eigs[-1]

# Do the same experiment again
v_in = np.random.randn(k, 1)

norm_list = [np.linalg.norm(v_in)]
for i in range(1, 100):
```

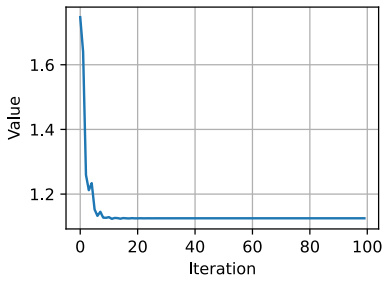
(continues on next page)



(continued from previous page)

```
    v_in = A.dot(v_in)
    norm_list.append(np.linalg.norm(v_in))

d2l.plot(np.arange(0, 100), norm_list, 'Iteration', 'Value')
```




```
# Rescale the matrix `A`
A /= norm_eigs[-1]

# Do the same experiment again
v_in = tf.random.normal((k, 1), dtype=tf.float64)

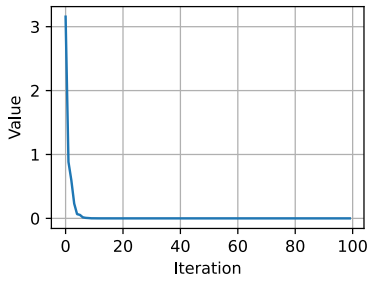
norm_list = [tf.norm(v_in).numpy()]
for i in range(1, 100):
    v_in = tf.matmul(A, v_in)
    norm_list.append(tf.norm(v_in).numpy())

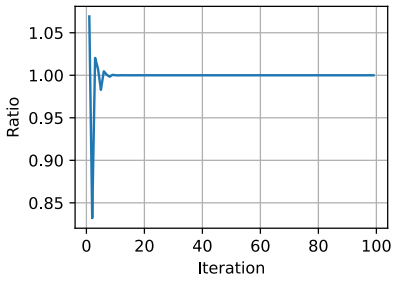
d2l.plot(tf.range(0, 100), norm_list, 'Iteration', 'Value')
```

前と同様に連続するノルムの比も描けば、実際に安定していくことがわかる。

```
# Also plot the ratio
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i-1])

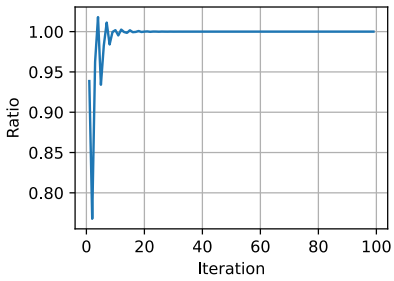
d2l.plot(torch.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```






```
# Also plot the ratio
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i-1])

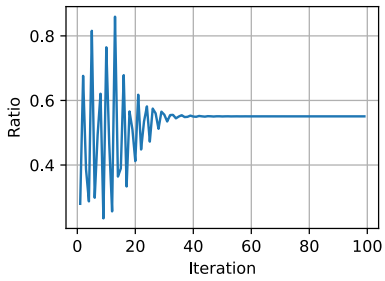
d2l.plot(np.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```




```
# Also plot the ratio
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i-1])

d2l.plot(tf.range(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```

25.2.7 議論



これで、私たちが望んでいたことがまさに起きているのがわかる！主固有値で行列を正規化すると、ランダムデータは以前のように爆発せず、最終的にはある特定の値に落ち着きる。これらを第一原理から導けるとよいのであるが、実際にその数学を深く見ると、独立な平均 0、分散 1 のガウス成分をもつ大きなランダム行列の最大固有値は、平均するとおよそ $\sqrt{n}$、この場合は $\sqrt{5} \approx 2.2$ になることがわかる。これは 円法則として知られる興味深い事実によるものです (Ginibre, 1965)。ランダム行列の固有値（およびそれに関連する特異値）と適切なニューラルネットワーク初期化との関係は、Pennington *et al.* (2017) およびその後の研究で深い結びつきがあることが示されている。

25.2.8 まとめ

- 固有ベクトルとは、行列によって方向を変えずに伸ばされるベクトルである。
- 固有値とは、行列の作用によって固有ベクトルがどれだけ伸ばされるかを表す量である。
- 行列の固有分解により、多くの演算を固有値に対する演算へと還元できる。
- ガーシュゴリンの円定理は、行列の固有値の近似値を与えてくれる。
- 反復的な行列べきの振る舞いは、主に最大固有値の大きさに依存する。この理解はニューラルネットワーク初期化の理論に多くの応用をもつ。

25.2.9 演習

1. 次の行列の固有値と固有ベクトルは何か。

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}? \tag{25.2.21}$$

2. 次の行列の固有値と固有ベクトルは何か。また、前の例と比べてこの例の何が奇妙か。

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}. \tag{25.2.22}$$

3. 固有値を計算せずに、次の行列の最小固有値が 0.5 より小さい可能性はあるか。注: この問題は頭の中で解ける。

$$\mathbf{A} = \begin{bmatrix} 3.0 & 0.1 & 0.3 & 1.0 \\ 0.1 & 1.0 & 0.1 & 0.2 \\ 0.3 & 0.1 & 5.0 & 0.0 \\ 1.0 & 0.2 & 0.0 & 1.8 \end{bmatrix}. \tag{25.2.23}$$

25.3 単変数微積分

Section 5.4 では、微分積分学の基本要素を見た。この節では、微積分の基礎と、それを機械学習の文脈でどのように理解し適用できるかを、さらに深く掘り下げる。

25.3.1 微分積分学

微分積分学は本質的に、関数が小さな変化の下でどのように振る舞うかを研究する学問である。これが深層学習にとってなぜ核心的なのかを見るために、例を考えてみよう。

便利のため、重みが単一のベクトル $\mathbf{w} = (w_1, \ldots, w_n)$ に連結されている深層ニューラルネットワークがあるとする。訓練データセットが与えられたとき、このデータセット上でのニューラルネットワークの損失を考える。これを $\mathcal{L}(\mathbf{w})$ と書くことにする。

この関数は非常に複雑で、与えられたアーキテクチャのすべての可能なモデルがこのデータセット上でどの程度性能を発揮するかを符号化しているため、どの重みの集合 $\mathbf{w}$ が損失を最小化するのかを見極めるのはほとんど不可能である。したがって実際には、重みを ランダムに初期化し、その後、損失をできるだけ速く減少させる方向へ小さな一歩を反復的に進めることがよくある。

すると問題は、一見するとそれほど簡単ではないものになる。すなわち、重みを最も速く減少させる方向をどう見つけるか、である。これを掘り下げるために、まず単一の重みだけの場合、すなわち単一の実数値 x に対して $L(\mathbf{w}) = L(x)$ となる場合を考えよう。

x を取り、それを小さく $x + \epsilon$ に変えたときに何が起こるかを理解してみよう。具体的な値を思い浮かべたいなら、$\epsilon = 0.0000001$ のような数を考えてみる。何が起こるかを視覚化するために、例として関数 $f(x) = \sin(x^x)$ を $[0, 3]$ 上で描いてみよう。

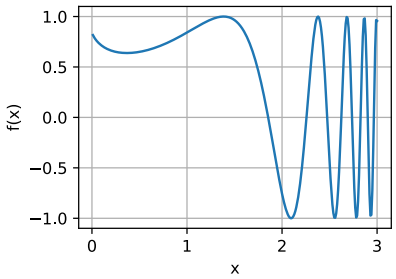
```
%matplotlib inline
from d2l import torch as d2l
```

(continues on next page)

(continued from previous page)

```
from IPython import display
import torch
torch.pi = torch.acos(torch.zeros(1)).item() * 2  # Define pi in torch

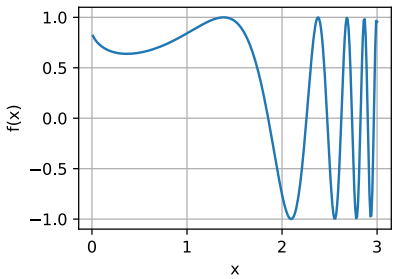
# Plot a function in a normal range
x_big = torch.arange(0.01, 3.01, 0.01)
ys = torch.sin(x_big**x_big)
d2l.plot(x_big, ys, 'x', 'f(x)')
```




```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from mxnet import np, npx
npx.set_np()

# Plot a function in a normal range
x_big = np.arange(0.01, 3.01, 0.01)
ys = np.sin(x_big**x_big)
d2l.plot(x_big, ys, 'x', 'f(x)')
```

```
[07:06:40] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```




```
# Define our function
def L(x):
    return x**2 + 1701*(x-4)**3
```

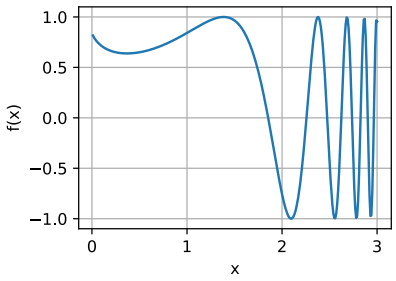
(continues on next page)

(continued from previous page)

```
# Print the difference divided by epsilon for several epsilon
for epsilon in [0.1, 0.001, 0.0001, 0.00001]:
    print(f'epsilon = {epsilon:.5f} -> {(L(4+epsilon) - L(4)) / epsilon:.5f}')
```

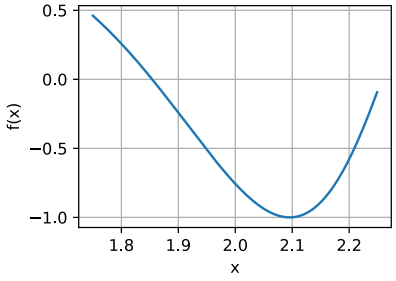
```
%matplotlib inline
from d2l import tensorflow as d2l
from IPython import display
import tensorflow as tf
tf.pi = tf.acos(tf.zeros(1)).numpy() * 2  # Define pi in TensorFlow

# Plot a function in a normal range
x_big = tf.range(0.01, 3.01, 0.01)
ys = tf.sin(x_big**x_big)
d2l.plot(x_big, ys, 'x', 'f(x)')
```

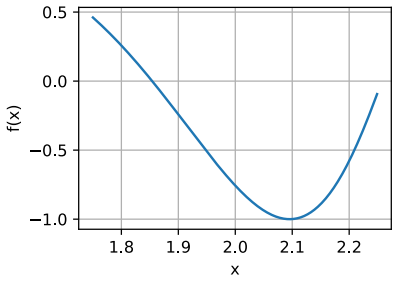


この大きなスケールでは、関数の振る舞いは単純ではない。しかし、範囲を $[1.75, 2.25]$ のようにもっと小さくすると、グラフがずっと単純になることがわかる。

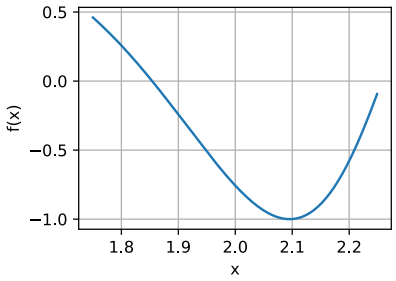
```
# Plot a the same function in a tiny range
x_med = torch.arange(1.75, 2.25, 0.001)
ys = torch.sin(x_med**x_med)
d2l.plot(x_med, ys, 'x', 'f(x)')
```



```
# Plot a the same function in a tiny range
x_med = np.arange(1.75, 2.25, 0.001)
ys = np.sin(x_med**x_med)
d2l.plot(x_med, ys, 'x', 'f(x)')
```




```
# Plot a the same function in a tiny range
x_med = tf.range(1.75, 2.25, 0.001)
ys = tf.sin(x_med**x_med)
d2l.plot(x_med, ys, 'x', 'f(x)')
```



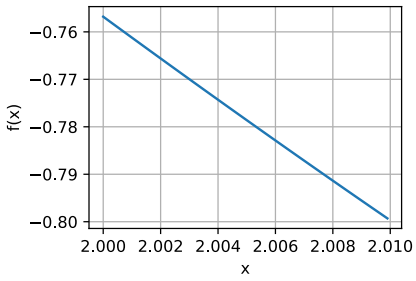
さらに極端に、非常に小さな区間まで拡大すると、振る舞いはさらに単純になり、ただの直線になる。

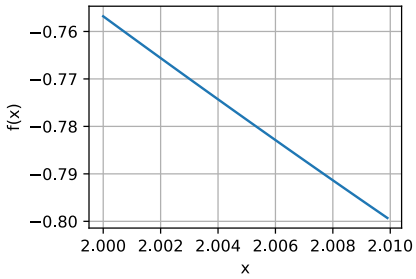
```
# Plot a the same function in a tiny range
x_small = torch.arange(2.0, 2.01, 0.0001)
ys = torch.sin(x_small**x_small)
d2l.plot(x_small, ys, 'x', 'f(x)')
```

```
# Plot a the same function in a tiny range
x_small = np.arange(2.0, 2.01, 0.0001)
ys = np.sin(x_small**x_small)
d2l.plot(x_small, ys, 'x', 'f(x)')
```

```
# Plot a the same function in a tiny range
x_small = tf.range(2.0, 2.01, 0.0001)
```

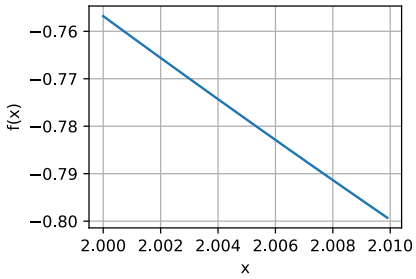
(continues on next page)





(continued from previous page)

```
ys = tf.sin(x_small**x_small)
d2l.plot(x_small, ys, 'x', 'f(x)')
```



これが単変数微積分の重要な観察である。よく知られた関数の振る舞いは、十分に小さな範囲では直線で近似できるのである。これは、ほとんどの関数について、x の値を少し動かすと出力 $f(x)$ も少し動くと期待するのが妥当であることを意味する。答えるべき唯一の問いは、「出力の変化は入力の変化に比べてどれくらい大きいのか。半分なのか、2 倍なのか？」である。

したがって、関数の入力を小さく変えたときの出力の変化量の比を考えることができる。これを形式的には次のように書ける。

$$\frac{L(x+\epsilon) - L(x)}{(x+\epsilon) - x} = \frac{L(x+\epsilon) - L(x)}{\epsilon}. \tag{25.3.1}$$

これは、コードで試してみるにはもう十分である。たとえば、$L(x) = x^2 +$

$1701(x-4)^3$ だとわかっているとしよう。このとき、$x=4$ におけるこの値の大きさを次のように調べられる。

```
# Define our function
def L(x):
    return x**2 + 1701*(x-4)**3

# Print the difference divided by epsilon for several epsilon
for epsilon in [0.1, 0.001, 0.0001, 0.00001]:
    print(f'epsilon = {epsilon:.5f} -> {(L(4+epsilon) - L(4)) / epsilon:.5f}')
```

```
epsilon = 0.10000 -> 25.11000
epsilon = 0.00100 -> 8.00270
epsilon = 0.00010 -> 8.00012
epsilon = 0.00001 -> 8.00001
```

ここで注意深く見ると、この数の出力が怪しいほど 8 に近いことに気づくだろう。実際、ϵ を小さくすると、値は次第に 8 に近づいていきる。したがって、求める値（入力の変化が出力をどれだけ変えるかの度合い）は、$x=4$ において 8 であると正しく結論できる。数学者はこの事実を次のように表する。

$$\lim_{\epsilon \to 0} \frac{L(4+\epsilon) - L(4)}{\epsilon} = 8. \tag{25.3.2}$$

少し歴史的な寄り道をすると、ニューラルネットワーク研究の最初の数十年間、科学者たちはこのアルゴリズム（有限差分法）を使って、損失関数が小さな摂動の下でどう変化するかを評価していた。つまり、重みを変えて損失がどう変わるかを見るのである。これは計算効率が悪く、1 つの変数の 1 つの変化が損失にどう影響するかを見るために損失関数を 2 回評価する必要がある。これを数千個程度のパラメータに対して行おうとすると、データセット全体に対するネットワークの評価が数千回も必要になる。1986 年になってようやく、Rumelhart *et al.* (1988) で導入された 誤差逆伝播アルゴリズムにより、重みの 任意の変化をまとめて考えたときに損失がどう変わるかを、データセットに対するネットワークの 1 回の予測と同じ計算時間で求める方法が与えられた。

先ほどの例に戻ると、この値 8 は x の値によって異なるので、x の関数として定義するのが自然である。より形式的には、この値に依存する変化率を 導関数と呼び、次のように書く。

$$\frac{df}{dx}(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}. \tag{25.3.3}$$

文献によって、導関数の表記はさまざまである。たとえば、以下の表記はすべて同じものを表している。

$$\frac{df}{dx} = \frac{d}{dx} f = f' = \nabla_x f = D_x f = f_x. \tag{25.3.4}$$

多くの著者は 1 つの表記を選んで使い続けるが、それすら必ずしも一貫しているとは限りない。これらすべてに慣れておくのがよいだろう。本書では、複雑な式の導関数を取りたい場合を除いて、$\frac{df}{dx}$ という表記を使う。その場合には、

次のような式を書くために $\frac{d}{dx}f$ を使う。

$$\frac{d}{dx}\left[x^4 + \cos\left(\frac{x^2+1}{2x-1}\right)\right]. \tag{25.3.5}$$

しばしば、導関数の定義 (25.3.3) をもう一度ほどいて、x を少し変えたときに関数がどう変わるかを直感的に見ると役立つ。

$$\begin{aligned} \frac{df}{dx}(x) = \lim_{\epsilon\to 0}\frac{f(x+\epsilon)-f(x)}{\epsilon} &\implies \frac{df}{dx}(x) \approx \frac{f(x+\epsilon)-f(x)}{\epsilon} \\ &\implies \epsilon\frac{df}{dx}(x) \approx f(x+\epsilon)-f(x) \\ &\implies f(x+\epsilon) \approx f(x) + \epsilon\frac{df}{dx}(x). \end{aligned} \tag{25.3.6}$$

最後の式は明示的に強調する価値がある。これは、任意の関数について、入力を少し変えると、出力はその小さな変化量に導関数を掛けた分だけ変わることを示している。

このように、導関数は、入力の変化に対して出力がどれだけ変化するかを教えてくれるスケーリング係数として理解できる。

25.3.2 微積分の規則

ここからは、明示的な関数の導関数をどう計算するかを理解する課題に移る。微積分を完全に厳密に扱うなら、すべてを第一原理から導くことになる。ここではその誘惑に乗るのではなく、よく出てくる規則を理解することにする。

よく使う導関数

Section 5.4 で見たように、導関数を計算するときには、しばしば一連の規則を使って計算をいくつかの基本関数に還元できる。参照しやすいように、ここで繰り返しておきる。

- **定数の導関数**。 $\frac{d}{dx}c = 0$。
- **線形関数の導関数**。 $\frac{d}{dx}(ax) = a$。
- **べき乗則**。 $\frac{d}{dx}x^n = nx^{n-1}$。
- **指数関数の導関数**。 $\frac{d}{dx}e^x = e^x$。
- **対数関数の導関数**。 $\frac{d}{dx}\log(x) = \frac{1}{x}$。

導関数の規則

必要な導関数をすべて個別に計算して表に保存しなければならないとしたら、微分積分学はほとんど不可能だろう。上の導関数を一般化し、たとえば $f(x) = \log\left(1+(x-1)^{10}\right)$ の導関数を求めるような、より複雑な導関数を計算できるのは数学の恩恵である。Section 5.4 で述べたように、その鍵は、関数をさまざ

まな方法で組み合わせたときに何が起こるか、特に和、積、合成を規則化することである。

- **和の法則**。 $\frac{d}{dx}\left(g(x)+h(x)\right)=\frac{dg}{dx}(x)+\frac{dh}{dx}(x)$。
- **積の法則**。 $\frac{d}{dx}\left(g(x)\cdot h(x)\right)=g(x)\frac{dh}{dx}(x)+\frac{dg}{dx}(x)h(x)$。
- **連鎖律**。 $\frac{d}{dx}g(h(x))=\frac{dg}{dh}(h(x))\cdot\frac{dh}{dx}(x)$。

(25.3.6) を使って、これらの規則をどう理解できるか見てみよう。和の法則については、次の推論を考える。

$$\begin{aligned}f(x+\epsilon)&=g(x+\epsilon)+h(x+\epsilon)\\&\approx g(x)+\epsilon\frac{dg}{dx}(x)+h(x)+\epsilon\frac{dh}{dx}(x)\\&=g(x)+h(x)+\epsilon\left(\frac{dg}{dx}(x)+\frac{dh}{dx}(x)\right)\\&=f(x)+\epsilon\left(\frac{dg}{dx}(x)+\frac{dh}{dx}(x)\right).\end{aligned}\tag{25.3.7}$$

この結果を、$f(x+\epsilon)\approx f(x)+\epsilon\frac{df}{dx}(x)$ という事実と比べると、望みどおり $\frac{df}{dx}(x)=\frac{dg}{dx}(x)+\frac{dh}{dx}(x)$ であることがわかる。ここでの直感は、入力 x を変えると、g と h がそれぞれ $\frac{dg}{dx}(x)$ と $\frac{dh}{dx}(x)$ だけ出力の変化に寄与する、というものである。

積はより微妙で、これらの式を扱うための新しい観察が必要になる。まずはこれまでと同様に (25.3.6) を使って始めよう。

$$\begin{aligned}f(x+\epsilon)&=g(x+\epsilon)\cdot h(x+\epsilon)\\&\approx\left(g(x)+\epsilon\frac{dg}{dx}(x)\right)\cdot\left(h(x)+\epsilon\frac{dh}{dx}(x)\right)\\&=g(x)\cdot h(x)+\epsilon\left(g(x)\frac{dh}{dx}(x)+\frac{dg}{dx}(x)h(x)\right)+\epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x)\\&=f(x)+\epsilon\left(g(x)\frac{dh}{dx}(x)+\frac{dg}{dx}(x)h(x)\right)+\epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x).\end{aligned}\tag{25.3.8}$$

これは上で行った計算に似ているし、実際、答え（$\frac{df}{dx}(x)=g(x)\frac{dh}{dx}(x)+\frac{dg}{dx}(x)h(x)$）が ϵ の隣に現れている。しかし、ϵ^2 の大きさを持つ項があるのが問題である。これを 高次項と呼ぶ。なぜなら、ϵ^2 のべきは ϵ^1 のべきより高いからである。後の節で、これらを追跡したい場合があることを見るが、今は $\epsilon=0.0000001$ なら $\epsilon^2=0.0000000000001$ であり、はるかに小さいことに注意する。$\epsilon\to 0$ とすると、高次項は安全に無視できる。この付録では一般的な慣習として、2 つの項が高次項を除いて等しいことを示すのに“≈”を使う。ただし、より厳密にしたいなら、差分商を調べればよいだろう。

$$\frac{f(x+\epsilon)-f(x)}{\epsilon}=g(x)\frac{dh}{dx}(x)+\frac{dg}{dx}(x)h(x)+\epsilon\frac{dg}{dx}(x)\frac{dh}{dx}(x),\tag{25.3.9}$$

そして $\epsilon\to 0$ とすると、右辺の最後の項も 0 に収束することがわかる。

最後に、連鎖律についても、再び (25.3.6) を使って進めると、

$$\begin{aligned} f(x+\epsilon) &= g(h(x+\epsilon)) \\ &\approx g\left(h(x)+\epsilon\frac{dh}{dx}(x)\right) \\ &\approx g(h(x))+\epsilon\frac{dh}{dx}(x)\frac{dg}{dh}(h(x)) \\ &= f(x)+\epsilon\frac{dg}{dh}(h(x))\frac{dh}{dx}(x), \end{aligned} \tag{25.3.10}$$

となる。ここで 2 行目では、関数 g の入力が（$h(x)$ から）微小量 $\epsilon\frac{dh}{dx}(x)$ だけずれたものとして見ている。

これらの規則により、実質的にどんな式でも計算できる柔軟な道具立てが得られる。たとえば、

$$\begin{aligned} \frac{d}{dx}\left[\log\left(1+(x-1)^{10}\right)\right] &= \left(1+(x-1)^{10}\right)^{-1}\frac{d}{dx}\left[1+(x-1)^{10}\right] \\ &= \left(1+(x-1)^{10}\right)^{-1}\left(\frac{d}{dx}[1]+\frac{d}{dx}[(x-1)^{10}]\right) \\ &= \left(1+(x-1)^{10}\right)^{-1}\left(0+10(x-1)^{9}\frac{d}{dx}[x-1]\right) \\ &= 10\left(1+(x-1)^{10}\right)^{-1}(x-1)^{9} \\ &= \frac{10(x-1)^{9}}{1+(x-1)^{10}}. \end{aligned} \tag{25.3.11}$$

各行では次の規則を使っている。

1. 連鎖律と対数関数の導関数。

2. 和の法則。

3. 定数の導関数、連鎖律、べき乗則。

4. 和の法則、線形関数の導関数、定数の導関数。

この例を通して、2 つのことが明らかになるはずである。

1. 和、積、定数、べき乗、指数関数、対数関数を使って書ける任意の関数は、これらの規則に従うことで機械的に導関数を計算できる。

2. 人間がこれらの規則に従うのは面倒で、誤りも起こりやすい！

ありがたいことに、この 2 つの事実を合わせると、先へ進む道が見えてきる。これは機械化のための完璧な候補である。実際、この節の後半で再び取り上げる誤差逆伝播は、まさにそれなのである。

線形近似

導関数を扱うとき、上で使った近似を幾何学的に解釈するとしばしば有用である。特に、次の式に注意する。

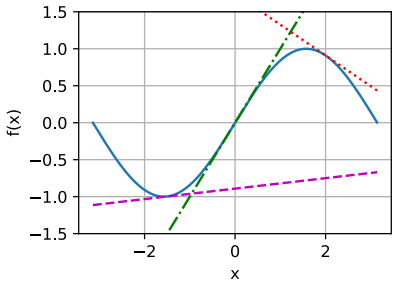
$$f(x+\epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x), \tag{25.3.12}$$

これは、点 $(x, f(x))$ を通り、傾きが $\frac{df}{dx}(x)$ の直線で f の値を近似している。このように、導関数は関数 f の線形近似を与えると言える。以下にその例を示す。

```
# Compute sin
xs = torch.arange(-torch.pi, torch.pi, 0.01)
plots = [torch.sin(xs)]

# Compute some linear approximations. Use d(sin(x))/dx = cos(x)
for x0 in [-1.5, 0.0, 2.0]:
    plots.append(torch.sin(torch.tensor(x0)) + (xs - x0) *
                 torch.cos(torch.tensor(x0)))

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```




```
# Compute sin
xs = np.arange(-np.pi, np.pi, 0.01)
plots = [np.sin(xs)]

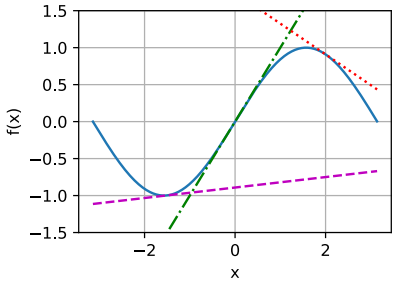
# Compute some linear approximations. Use d(sin(x)) / dx = cos(x)
for x0 in [-1.5, 0, 2]:
    plots.append(np.sin(x0) + (xs - x0) * np.cos(x0))

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```

```
# Compute sin
xs = tf.range(-tf.pi, tf.pi, 0.01)
plots = [tf.sin(xs)]

# Compute some linear approximations. Use d(sin(x))/dx = cos(x)
for x0 in [-1.5, 0.0, 2.0]:
    plots.append(tf.sin(tf.constant(x0)) + (xs - x0) *
```

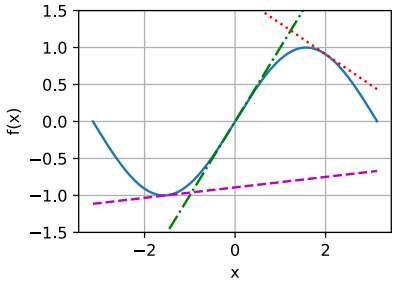
(continues on next page)



(continued from previous page)

```
                tf.cos(tf.constant(x0)))

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



高階導関数

ここで、表面的には奇妙に見えるかもしれないことをしてみよう。関数 f を取り、その導関数 $\frac{df}{dx}$ を計算する。これにより、任意の点での f の変化率が得られる。

しかし、導関数 $\frac{df}{dx}$ 自体も関数と見なせるので、$\frac{df}{dx}$ の導関数を計算して $\frac{d^2f}{dx^2} = \frac{df}{dx}\left(\frac{df}{dx}\right)$ を得ることを妨げるものはない。これを f の 2 階導関数と呼ぶ。この関数は、f の変化率の変化率、言い換えれば、変化率がどのように変化しているかを表する。導関数は何回でも適用でき、n 階導関数と呼ばれるものが得られる。記法をすっきりさせるため、n 階導関数を次のように表する。

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{25.3.13}$$

これがなぜ有用な概念なのかを理解してみよう。以下では、$f^{(2)}(x)$、$f^{(1)}(x)$、および $f(x)$ を可視化する。

まず、2 階導関数 $f^{(2)}(x)$ が正の定数である場合を考える。これは、1 階導関数の傾きが正であることを意味する。その結果、1 階導関数 $f^{(1)}(x)$ は最初は負で、ある点で 0 になり、最後には正になる。これは元の関数 f の傾きを示している

ので、関数 f 自体は減少し、平らになり、その後増加する。言い換えれば、関数 f は上に曲がっており、Fig. 25.3.1 に示すように 1 つの最小値を持つ。

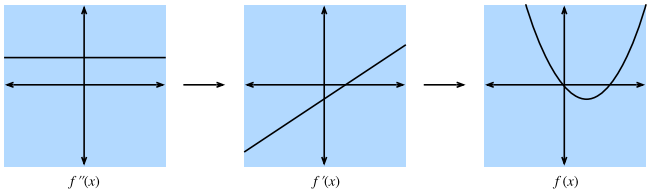


Fig. 25.3.1 2 階導関数が正の定数だと仮定すると、1 階導関数は増加し、それにより関数自体は最小値を持つ。

次に、2 階導関数が負の定数である場合、それは 1 階導関数が減少していることを意味する。これは、1 階導関数が最初は正で、ある点で 0 になり、その後負になることを示す。したがって、関数 f 自体は増加し、平らになり、その後減少する。言い換えれば、関数 f は下に曲がっており、Fig. 25.3.2 に示すように 1 つの最大値を持つ。

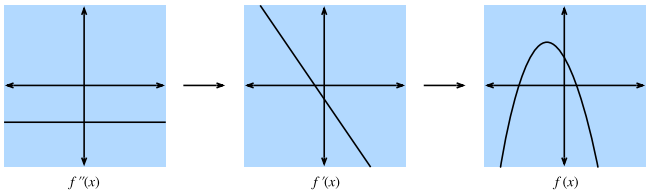


Fig. 25.3.2 2 階導関数が負の定数だと仮定すると、1 階導関数は減少し、それにより関数自体は最大値を持つ。

第三に、2 階導関数が常に 0 であれば、1 階導関数は決して変化せず、一定です！これは、f が一定の速度で増加（または減少）し、f 自体が Fig. 25.3.3 に示すように直線であることを意味する。

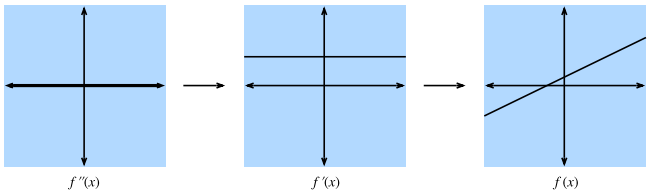


Fig. 25.3.3 2 階導関数が 0 だと仮定すると、1 階導関数は一定であり、それにより関数自体は直線になる。

要するに、2 階導関数は関数 f の曲がり方を表していると解釈できる。正の 2 階導関数は上向きの曲がりをもたらし、負の 2 階導関数は下向きの曲がりを意味し、0 の 2 階導関数は関数 f がまったく曲がっていないことを意味する。

これをもう一歩進めよう。関数 $g(x) = ax^2 + bx + c$ を考える。このとき、

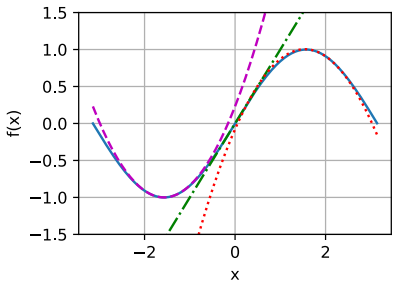
$$\begin{aligned}\frac{dg}{dx}(x) &= 2ax + b \\ \frac{d^2g}{dx^2}(x) &= 2a.\end{aligned} \tag{25.3.14}$$

もし元の関数 $f(x)$ があるなら、最初の 2 つの導関数を計算し、それらに一致するような a, b, c の値を見つけることができる。前の節で、1 階導関数が直線による最良近似を与えることを見たが、同様に、この構成は 2 次式による最良近似を与える。これを $f(x) = \sin(x)$ で可視化してみよう。

```
# Compute sin
xs = torch.arange(-torch.pi, torch.pi, 0.01)
plots = [torch.sin(xs)]

# Compute some quadratic approximations. Use d(sin(x)) / dx = cos(x)
for x0 in [-1.5, 0.0, 2.0]:
    plots.append(torch.sin(torch.tensor(x0)) + (xs - x0) *
                 torch.cos(torch.tensor(x0)) - (xs - x0)**2 *
                 torch.sin(torch.tensor(x0)) / 2)

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```




```
# Compute sin
xs = np.arange(-np.pi, np.pi, 0.01)
plots = [np.sin(xs)]

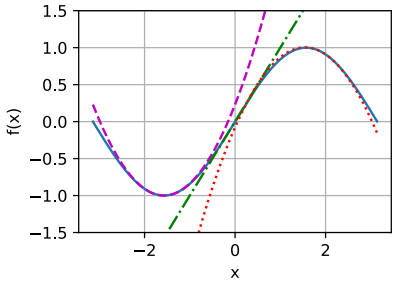
# Compute some quadratic approximations. Use d(sin(x)) / dx = cos(x)
for x0 in [-1.5, 0, 2]:
    plots.append(np.sin(x0) + (xs - x0) * np.cos(x0) -
                 (xs - x0)**2 * np.sin(x0) / 2)

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```

```
# Compute sin
xs = tf.range(-tf.pi, tf.pi, 0.01)
plots = [tf.sin(xs)]

# Compute some quadratic approximations. Use d(sin(x)) / dx = cos(x)
```

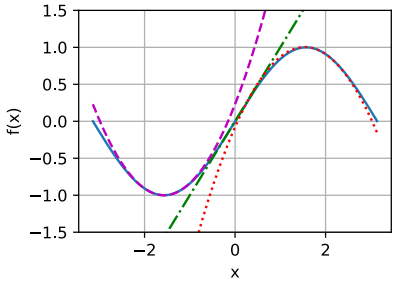
(continues on next page)



(continued from previous page)

```
for x0 in [-1.5, 0.0, 2.0]:
    plots.append(tf.sin(tf.constant(x0)) + (xs - x0) *
                 tf.cos(tf.constant(x0)) - (xs - x0)**2 *
                 tf.sin(tf.constant(x0)) / 2)

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



この考え方は次の節で テイラー級数に拡張する。

テイラー級数

テイラー級数は、点x_0における最初のn個の導関数の値、すなわち$\{f(x_0), f^{(1)}(x_0), f^{(2)}(x_0), \ldots, f^{(n)}(x_0)$が与えられたときに、関数$f(x)$を近似する方法を与える。考え方は、$x_0$において与えられたすべての導関数に一致する$n$次多項式を見つけることである。

前節で$n = 2$の場合を見たが、少し代数計算をすると、これは次のようになる。

$$f(x) \approx \frac{1}{2}\frac{d^2 f}{dx^2}(x_0)(x - x_0)^2 + \frac{df}{dx}(x_0)(x - x_0) + f(x_0). \tag{25.3.15}$$

上で見たように、分母の2はx^2を2回微分したときに得られる2を打ち消すためにあり、他の項はすべて0になる。同じ論理が1階導関数と関数値そのものにも当てはまる。

この論理を $n = 3$ まで進めると、次を得る。

$$f(x) \approx \frac{\frac{d^3 f}{dx^3}(x_0)}{6}(x - x_0)^3 + \frac{\frac{d^2 f}{dx^2}(x_0)}{2}(x - x_0)^2 + \frac{df}{dx}(x_0)(x - x_0) + f(x_0). \tag{25.3.16}$$

ここで $6 = 3 \times 2 = 3!$ は、x^3 を 3 回微分したときに前に現れる定数に由来する。

さらに、次のようにして n 次多項式を得ることができる。

$$P_n(x) = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i. \tag{25.3.17}$$

ここでの記法

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{25.3.18}$$

実際、$P_n(x)$ は関数 $f(x)$ に対する最良の n 次多項式近似と見なせる。

上の近似の誤差を最後まで掘り下げることはしないが、無限極限について触れておく価値はある。この場合、$\cos(x)$ や e^x のような、うまく振る舞う関数（実解析関数として知られる）については、無限個の項を書き下して、まったく同じ関数を近似できる。

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n. \tag{25.3.19}$$

例として $f(x) = e^x$ を考えよう。e^x は自分自身の導関数なので、$f^{(n)}(x) = e^x$ である。したがって、$x_0 = 0$ でテイラー級数を取ることで、e^x は次のように再構成できる。

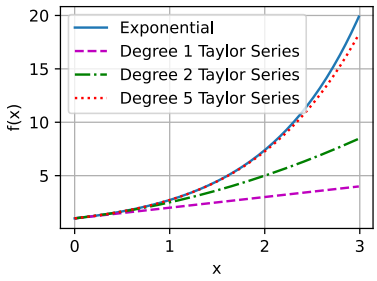
$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots. \tag{25.3.20}$$

これがコードではどうなるかを見て、テイラー近似の次数を上げると目的の関数 e^x にどんどん近づくことを観察しよう。

```
# Compute the exponential function
xs = torch.arange(0, 3, 0.01)
ys = torch.exp(xs)

# Compute a few Taylor series approximations
P1 = 1 + xs
P2 = 1 + xs + xs**2 / 2
P5 = 1 + xs + xs**2 / 2 + xs**3 / 6 + xs**4 / 24 + xs**5 / 120

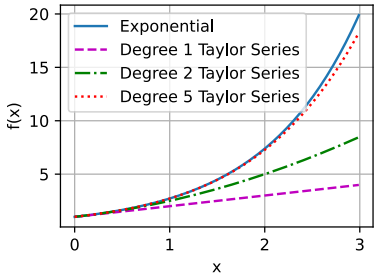
d2l.plot(xs, [ys, P1, P2, P5], 'x', 'f(x)', legend=[
    "Exponential", "Degree 1 Taylor Series", "Degree 2 Taylor Series",
    "Degree 5 Taylor Series"])
```




```
# Compute the exponential function
xs = np.arange(0, 3, 0.01)
ys = np.exp(xs)

# Compute a few Taylor series approximations
P1 = 1 + xs
P2 = 1 + xs + xs**2 / 2
P5 = 1 + xs + xs**2 / 2 + xs**3 / 6 + xs**4 / 24 + xs**5 / 120

d2l.plot(xs, [ys, P1, P2, P5], 'x', 'f(x)', legend=[
    "Exponential", "Degree 1 Taylor Series", "Degree 2 Taylor Series",
    "Degree 5 Taylor Series"])
```

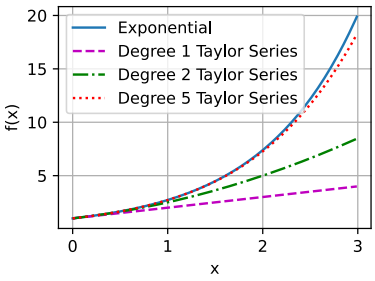



```
# Compute the exponential function
xs = tf.range(0, 3, 0.01)
ys = tf.exp(xs)

# Compute a few Taylor series approximations
P1 = 1 + xs
P2 = 1 + xs + xs**2 / 2
P5 = 1 + xs + xs**2 / 2 + xs**3 / 6 + xs**4 / 24 + xs**5 / 120

d2l.plot(xs, [ys, P1, P2, P5], 'x', 'f(x)', legend=[
    "Exponential", "Degree 1 Taylor Series", "Degree 2 Taylor Series",
    "Degree 5 Taylor Series"])
```

テイラー級数には主に 2 つの応用がある。



1. 理論的応用: 複雑すぎる関数を理解しようとするとき、テイラー級数を使うと、それを直接扱える多項式に変換できる。

2. 数値的応用: e^x や $\cos(x)$ のような関数は、機械にとって計算が難しいことがある。機械は固定精度で値の表を保持できる（実際そうすることも多い）が、それでも「$\cos(1)$ の 1000 桁目は何か？」のような問いは残る。テイラー級数は、こうした問いに答えるのにしばしば役立つ。

25.3.3 まとめ

- 導関数は、入力を少し変えたときに関数がどう変化するかを表すのに使える。

- 基本的な導関数は、導関数の規則を使って組み合わせることで、任意に複雑な導関数を作れる。

- 導関数は繰り返し適用でき、2 階以上の導関数を得られる。次数が上がるほど、関数の振る舞いについてより細かな情報が得られる。

- 単一のデータ例における導関数の情報を使うと、テイラー級数から得られる多項式によって、うまく振る舞う関数を近似できる。

25.3.4 演習

1. $x^3 - 4x + 1$ の導関数は何か？

2. $\log(\frac{1}{x})$ の導関数は何か？

3. 真か偽か: $f'(x) = 0$ ならば、f は x において最大値または最小値を持つか？

4. $x \geq 0$ に対する $f(x) = x\log(x)$ の最小値はどこか（ここで $f(0)$ では 0 の極限値を取ると仮定する）？

25.4 多変数微積分

単一変数関数の導関数についてかなり深く理解できたので、今度は、潜在的に何十億もの重みをもつ損失関数を考えていた元の問いに戻りよう。

25.4.1 高次元での微分

Section 25.3 が教えてくれるのは、他のすべてを固定したままこれら何十億もの重みのうちの 1 つだけを変えたとき、何が起こるかが分かるということだ！これは単なる 1 変数関数にほかならないので、次のように書ける。

$$L(w_1 + \epsilon_1, w_2, \ldots, w_N) \approx L(w_1, w_2, \ldots, w_N) + \epsilon_1 \frac{d}{dw_1} L(w_1, w_2, \ldots, w_N). \quad (25.4.1)$$

他の変数を固定したまま 1 つの変数についてとる導関数を 偏微分と呼び、(25.4.1) の導関数には $\frac{\partial}{\partial w_1}$ という記法を用いる。

では、これを使って w_2 を少しだけ $w_2 + \epsilon_2$ に変えてみよう。

$$\begin{aligned}
L(w_1 + \epsilon_1, w_2 + \epsilon_2, \ldots, w_N) &\approx L(w_1, w_2 + \epsilon_2, \ldots, w_N) + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2 + \epsilon_2, \ldots, w_N + \epsilon_N) \\
&\approx L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_2 \frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \epsilon_2 \frac{\partial}{\partial w_2} \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\
&\approx L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_2 \frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N).
\end{aligned} \quad (25.4.2)$$

ここでも、$\epsilon_1 \epsilon_2$ は高次の項であり、前節で ϵ^2 を捨てたのと同じように無視できる、という考え方を使いた。(25.4.1) で見たことも同様である。このように続けていくと、次のように書ける。

$$L(w_1 + \epsilon_1, w_2 + \epsilon_2, \ldots, w_N + \epsilon_N) \approx L(w_1, w_2, \ldots, w_N) + \sum_i \epsilon_i \frac{\partial}{\partial w_i} L(w_1, w_2, \ldots, w_N). \quad (25.4.3)$$

これは一見ごちゃごちゃして見えるが、右辺の和がちょうど内積の形になっていることに気づけば、もっと見慣れた形にできる。そこで

$$\boldsymbol{\epsilon} = [\epsilon_1, \ldots, \epsilon_N]^\top \text{ and } \nabla_{\mathbf{x}} L = \left[\frac{\partial L}{\partial x_1}, \ldots, \frac{\partial L}{\partial x_N} \right]^\top, \quad (25.4.4)$$

とおくと、

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{25.4.5}$$

ベクトル $\nabla_{\mathbf{w}} L$ を L の 勾配と呼ぶ。

式 (25.4.5) は少し立ち止まって考える価値がある。これは 1 次元で見たものとまったく同じ形式で、ただすべてをベクトルと内積に置き換えただけである。これにより、入力にどのような摂動を与えたときに関数 L がどのように変化するかを近似的に知ることができる。次節で見るように、これは勾配に含まれる情報を使って、学習が幾何学的にどのように行えるかを理解するうえで重要な道具になる。

しかしその前に、この近似が実際にどう働くかを例で見てみよう。次の関数を扱っているとする。

$$f(x, y) = \log(e^x + e^y) \text{ with gradient } \nabla f(x, y) = \left[\frac{e^x}{e^x + e^y}, \frac{e^y}{e^x + e^y}\right]. \tag{25.4.6}$$

$(0, \log(2))$ のような点を見ると、

$$f(x, y) = \log(3) \text{ with gradient } \nabla f(x, y) = \left[\frac{1}{3}, \frac{2}{3}\right]. \tag{25.4.7}$$

したがって、$(\epsilon_1, \log(2) + \epsilon_2)$ における f を近似したいなら、(25.4.5) の具体例として次が得られるはずである。

$$f(\epsilon_1, \log(2) + \epsilon_2) \approx \log(3) + \frac{1}{3}\epsilon_1 + \frac{2}{3}\epsilon_2. \tag{25.4.8}$$

これをコードで確かめて、近似がどれくらい良いか見てみよう。

```
%matplotlib inline
from d2l import torch as d2l
from IPython import display
from mpl_toolkits import mplot3d
import torch
import numpy as np

def f(x, y):
    return torch.log(torch.exp(x) + torch.exp(y))
def grad_f(x, y):
    return torch.tensor([torch.exp(x) / (torch.exp(x) + torch.exp(y)),
                         torch.exp(y) / (torch.exp(x) + torch.exp(y))])

epsilon = torch.tensor([0.01, -0.03])
grad_approx = f(torch.tensor([0.]), torch.log(
    torch.tensor([2.]))) + epsilon.dot(
    grad_f(torch.tensor([0.]), torch.log(torch.tensor(2.))))
true_value = f(torch.tensor([0.]) + epsilon[0], torch.log(
    torch.tensor([2.])) + epsilon[1])
f'approximation: {grad_approx}, true Value: {true_value}'
```

```
'approximation: tensor([1.0819]), true Value: tensor([1.0821])'
```

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from mpl_toolkits import mplot3d
from mxnet import autograd, np, npx
npx.set_np()

def f(x, y):
    return np.log(np.exp(x) + np.exp(y))
def grad_f(x, y):
    return np.array([np.exp(x) / (np.exp(x) + np.exp(y)),
                     np.exp(y) / (np.exp(x) + np.exp(y))])

epsilon = np.array([0.01, -0.03])
grad_approx = f(0, np.log(2)) + epsilon.dot(grad_f(0, np.log(2)))
true_value = f(0 + epsilon[0], np.log(2) + epsilon[1])
f'approximation: {grad_approx}, true Value: {true_value}'
```

```
[07:20:46] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
'approximation: 1.0819456577301025, true Value: 1.0821242332458496'
```

```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print(f'    f at {w}, {x}, {y}, {z} is {f}')

# Compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)

# Compute the final result from inputs to outputs
du_dw, dv_dw = du_da*da_dw + du_db*db_dw, dv_da*da_dw + dv_db*db_dw
df_dw = df_du*du_dw + df_dv*dv_dw
print(f'df/dw at {w}, {x}, {y}, {z} is {df_dw}')
```

```
%matplotlib inline
from d2l import tensorflow as d2l
from IPython import display
from mpl_toolkits import mplot3d
import tensorflow as tf
import numpy as np

def f(x, y):
    return tf.math.log(tf.exp(x) + tf.exp(y))
def grad_f(x, y):
    return tf.constant([(tf.exp(x) / (tf.exp(x) + tf.exp(y))).numpy(),
```

(continues on next page)

(continued from previous page)

```
                        (tf.exp(y) / (tf.exp(x) + tf.exp(y))).numpy()])

epsilon = tf.constant([0.01, -0.03])
grad_approx = f(tf.constant([0.]), tf.math.log(
    tf.constant([2.]))) + tf.tensordot(
    epsilon, grad_f(tf.constant([0.]), tf.math.log(tf.constant(2.))), axes=1)
true_value = f(tf.constant([0.]) + epsilon[0], tf.math.log(
    tf.constant([2.])) + epsilon[1])
f'approximation: {grad_approx}, true Value: {true_value}'
```

```
'approximation: [1.0819457], true Value: [1.0821242]'
```

25.4.2 勾配と勾配降下法の幾何学

(25.4.5) の式をもう一度考えましょう。

$$L(\mathbf{w}+\boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{25.4.9}$$

これを使って損失 L を最小化したいとする。まず、Section 5.5 で最初に説明した勾配降下法を幾何学的に理解しよう。やることは次の通りである。

1. 初期パラメータ $\mathbf{w}$ をランダムに選ぶ。
2. $\mathbf{w}$ において L を最も急速に減少させる方向 $\mathbf{v}$ を見つける。
3. その方向に少し進む: $\mathbf{w} \to \mathbf{w} + \epsilon\mathbf{v}$。
4. 繰り返す。

正確なやり方が分からないのは、第 2 段階でベクトル $\mathbf{v}$ をどう計算するかだけである。このような方向を 最急降下方向と呼ぶ。Section 25.1 での内積の幾何学的理解を使うと、(25.4.5) は次のように書き換えられる。

$$L(\mathbf{w}+\mathbf{v}) \approx L(\mathbf{w}) + \mathbf{v} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}) = L(\mathbf{w}) + \|\nabla_{\mathbf{w}} L(\mathbf{w})\| \cos(\theta). \tag{25.4.10}$$

ここでは、便宜上、方向ベクトルの長さを 1 にしてあり、θ は $\mathbf{v}$ と $\nabla_{\mathbf{w}} L(\mathbf{w})$ のなす角である。L をできるだけ急速に減少させる方向を見つけたいなら、この式をできるだけ負にしたいわけである。この式に方向の選び方が入ってくるのは $\cos(\theta)$ を通してだけなので、この余弦をできるだけ負にしたいことになる。ここで余弦の形を思い出すと、$\cos(\theta) = -1$ にすれば最も負にでき、言い換えれば勾配と選んだ方向のなす角を π ラジアン、つまり 180 度にすればよいことが分かる。これを達成する唯一の方法は、まったく逆向きに進むことである。つまり、$\mathbf{v}$ を $\nabla_{\mathbf{w}} L(\mathbf{w})$ と正反対の方向に向ければよいのです！

ここで、機械学習における最も重要な数学的概念の 1 つにたどり着きる。最急降下方向は $-\nabla_{\mathbf{w}} L(\mathbf{w})$ の方向を向きる。したがって、先ほどの直感的なアルゴリズムは次のように書き換えられる。

1. 初期パラメータ $\mathbf{w}$ をランダムに選ぶ。

2. $\nabla_{\mathbf{w}} L(\mathbf{w})$ を計算する。

3. その反対方向に少し進む: $\mathbf{w} \leftarrow \mathbf{w} - \epsilon \nabla_{\mathbf{w}} L(\mathbf{w})$。

4. 繰り返す。

この基本アルゴリズムは多くの研究者によってさまざまに修正・拡張されてきたが、核となる考え方はどれも同じである。勾配を使って損失をできるだけ急速に減少させる方向を見つけ、その方向に一歩進むようにパラメータを更新するのである。

25.4.3 数学的最適化についての注意

本書を通して、私たちは数値最適化手法に焦点を当てる。深層学習で出会う関数はどれも複雑すぎて、明示的に最小化することができないからである。

しかし、上で得た幾何学的理解が、関数を直接最適化することについて何を教えてくれるのかを考えるのは有益である。

ある関数 $L(\mathbf{x})$ を最小にする $\mathbf{x}_0$ を見つけたいとする。さらに、誰かがある値を与えて、それが L を最小にする値だと言ったとしよう。その答えがもっともらしいかどうかを確認する方法はあるだろうか？

もう一度 (25.4.5) を考える。

$$L(\mathbf{x}_0 + \boldsymbol{\epsilon}) \approx L(\mathbf{x}_0) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{x}} L(\mathbf{x}_0). \tag{25.4.11}$$

もし勾配がゼロでなければ、$-\epsilon \nabla_{\mathbf{x}} L(\mathbf{x}_0)$ の方向に一歩進むことで、より小さい L の値を見つけられることが分かる。したがって、本当に最小値にいるなら、これは起こりえない！ここから、$\mathbf{x}_0$ が最小値なら $\nabla_{\mathbf{x}} L(\mathbf{x}_0) = 0$ でなければならないと結論できる。$\nabla_{\mathbf{x}} L(\mathbf{x}_0) = 0$ を満たす点を 臨界点と呼ぶ。

これは便利である。というのも、まれな場合ではあるが、勾配がゼロになる点をすべて明示的に見つけ、その中で最小の値を与えるものを探せることがあるからである。

具体例として、次の関数を考える。

$$f(x) = 3x^4 - 4x^3 - 12x^2. \tag{25.4.12}$$

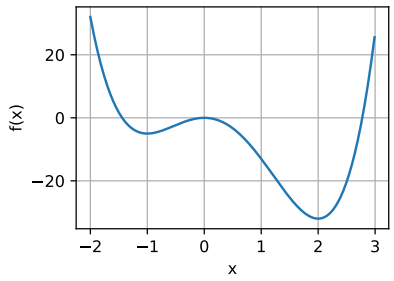
この関数の導関数は

$$\frac{df}{dx} = 12x^3 - 12x^2 - 24x = 12x(x-2)(x+1). \tag{25.4.13}$$

最小値の候補となる場所は $x = -1, 0, 2$ だけであり、それぞれの関数値は $-5, 0, -32$ である。したがって、$x = 2$ のときにこの関数が最小になると結論できる。簡単なプロットでこれを確認できる。

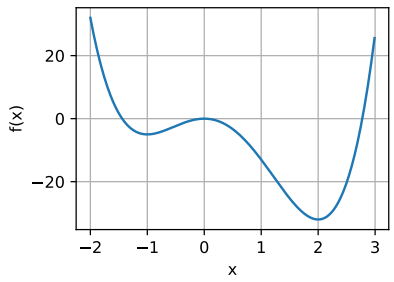
```
x = torch.arange(-2, 3, 0.01)
f = (3 * x**4) - (4 * x**3) - (12 * x**2)

d2l.plot(x, f, 'x', 'f(x)')
```




```
x = np.arange(-2, 3, 0.01)
f = (3 * x**4) - (4 * x**3) - (12 * x**2)

d2l.plot(x, f, 'x', 'f(x)')
```




```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print(f'f at {w}, {x}, {y}, {z} is {f}')

# Compute the derivative using the decomposition above
# First compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)
da_dx, db_dx = 2*(w + x + y + z), 2*(w + x - y - z)
da_dy, db_dy = 2*(w + x + y + z), -2*(w + x - y - z)
da_dz, db_dz = 2*(w + x + y + z), -2*(w + x - y - z)

# Now compute how f changes when we change any value from output to input
df_da, df_db = df_du*du_da + df_dv*dv_da, df_du*du_db + df_dv*dv_db
```

(continues on next page)

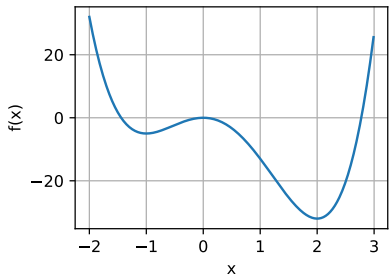
(continued from previous page)

```
df_dw, df_dx = df_da*da_dw + df_db*db_dw, df_da*da_dx + df_db*db_dx
df_dy, df_dz = df_da*da_dy + df_db*db_dy, df_da*da_dz + df_db*db_dz

print(f'df/dw at {w}, {x}, {y}, {z} is {df_dw}')
print(f'df/dx at {w}, {x}, {y}, {z} is {df_dx}')
print(f'df/dy at {w}, {x}, {y}, {z} is {df_dy}')
print(f'df/dz at {w}, {x}, {y}, {z} is {df_dz}')
```

```
x = tf.range(-2, 3, 0.01)
f = (3 * x**4) - (4 * x**3) - (12 * x**2)

d2l.plot(x, f, 'x', 'f(x)')
```



これは、理論的に扱う場合でも数値的に扱う場合でも知っておくべき重要な事実を示している。最小化（あるいは最大化）できる可能性がある点は勾配がゼロである必要があるが、勾配がゼロの点がすべて真の 大域的最小値（あるいは最大値）とは限りない。

25.4.4 多変数の連鎖律

4 変数 (w, x, y, z) の関数があり、それを多くの項を合成して作れるとしよう。

$$\begin{aligned} f(u, v) &= (u + v)^2 \\ u(a, b) &= (a + b)^2, \qquad v(a, b) = (a - b)^2, \\ a(w, x, y, z) &= (w + x + y + z)^2, \qquad b(w, x, y, z) = (w + x - y - z)^2. \end{aligned} \tag{25.4.14}$$

このような連鎖した方程式はニューラルネットワークを扱うときによく現れるので、そのような関数の勾配をどう計算するかを理解することが重要である。どの変数がどの変数に直接関係しているかを見ると、Fig. 25.4.1 にこの関係の視覚的な手がかりが見えてきる。

(25.4.14) のすべてをそのまま合成して

$$f(w, x, y, z) = \left(\left((w + x + y + z)^2 + (w + x - y - z)^2\right)^2 + \left((w + x + y + z)^2 - (w + x - y - z)^2\right)^2\right)^2. \tag{25.4.15}$$

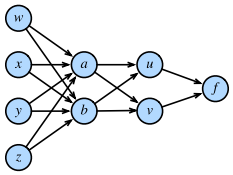


Fig. 25.4.1 上の関数関係。ノードは値を表し、辺は関数依存を示す。

と書くこともできる。

その後、単変数の導関数だけを使って微分することもできるが、そうするとすぐに項が大量に出てきて、その多くが重複していることに気づくだろう！実際、たとえば次のようになる。

$$\begin{aligned}\frac{\partial f}{\partial w} = 2\Big(2\left(2(w+x+y+z)-2(w+x-y-z)\right)\left((w+x+y+z)^2-(w+x-y-z)^2\right)+\\ 2\left(2(w+x-y-z)+2(w+x+y+z)\right)\left((w+x-y-z)^2+(w+x+y+z)^2\right)\Big)\times\\ \Big(\left((w+x+y+z)^2-(w+x-y-z)^2\right)^2+\left((w+x-y-z)^2+(w+x+y+z)^2\right)^2\Big).\end{aligned} \tag{25.4.16}$$

もし $\frac{\partial f}{\partial x}$ も計算したいなら、同様の式がまた現れ、多くの重複項、しかも 2 つの導関数の間で 共有された重複項がたくさん出てきる。これは膨大な無駄な計算を意味し、もしこのようなやり方で微分を計算しなければならなかったら、深層学習革命は始まる前に止まっていただろう！

問題を分解しよう。まず、a を少し変えたときに f がどう変わるかを理解することから始める。つまり、w, x, y, z は存在しないものとして考える。勾配を初めて扱ったときと同じように考えよう。a に小さな量 ϵ を加えてみる。

$$\begin{aligned}&f(u(a+\epsilon,b),v(a+\epsilon,b))\\ \approx& f\left(u(a,b)+\epsilon\frac{\partial u}{\partial a}(a,b), v(a,b)+\epsilon\frac{\partial v}{\partial a}(a,b)\right)\\ \approx& f(u(a,b),v(a,b))+\epsilon\left[\frac{\partial f}{\partial u}(u(a,b),v(a,b))\frac{\partial u}{\partial a}(a,b)+\frac{\partial f}{\partial v}(u(a,b),v(a,b))\frac{\partial v}{\partial a}(a,b)\right].\end{aligned} \tag{25.4.17}$$

1 行目は偏微分の定義から、2 行目は勾配の定義から従う。$\frac{\partial f}{\partial u}(u(a,b),v(a,b))$ のように、各導関数をどこで評価するかを正確に追うのは記法上やや煩雑なので、しばしば次のように簡潔に書く。

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a}+\frac{\partial f}{\partial v}\frac{\partial v}{\partial a}. \tag{25.4.18}$$

この過程の意味を考えると理解しやすくなる。私たちは $f(u(a,b),v(a,b))$ のような関数が、a の変化によってどのように値を変えるかを理解しようとしている。これが起こる経路は 2 つあり、$a \to u \to f$ の経路と $a \to v \to f$ の経路である。これら 2 つの寄与は連鎖律によってそれぞれ $\frac{\partial w}{\partial u}\cdot\frac{\partial u}{\partial x}$ と $\frac{\partial w}{\partial v}\cdot\frac{\partial v}{\partial x}$ として計算でき、それらを足し合わせる。

Fig. 25.4.2 に示すように、右側の関数が左側でつながっているものに依存する、別の関数ネットワークがあると想像する。

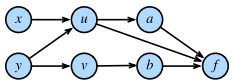


Fig. 25.4.2 連鎖律のもう少し繊細な例。

$\frac{\partial f}{\partial y}$ のようなものを計算するには、y から f へのすべての経路（この場合は 3 本）について和をとる必要があり、次のようになる。

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial v}\frac{\partial v}{\partial y}. \tag{25.4.19}$$

このように連鎖律を理解しておくことは、勾配がネットワークを通ってどのように流れるか、そして LSTM (Section 13.1) や残差層 (Section 11.6) のようなさまざまなアーキテクチャ上の選択が、勾配の流れを制御することで学習過程にどのように役立つかを理解するうえで大きな助けになる。

25.4.5 逆伝播アルゴリズム

前節の (25.4.14) の例に戻りよう。そこでは

$$\begin{aligned} f(u,v) &= (u+v)^2 \\ u(a,b) &= (a+b)^2, \qquad v(a,b) = (a-b)^2, \\ a(w,x,y,z) &= (w+x+y+z)^2, \qquad b(w,x,y,z) = (w+x-y-z)^2. \end{aligned} \tag{25.4.20}$$

たとえば $\frac{\partial f}{\partial w}$ を計算したいなら、多変数の連鎖律を適用して次のようにできる。

$$\begin{aligned} \frac{\partial f}{\partial w} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial w} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial w}, \\ \frac{\partial u}{\partial w} &= \frac{\partial u}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial u}{\partial b}\frac{\partial b}{\partial w}, \\ \frac{\partial v}{\partial w} &= \frac{\partial v}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial v}{\partial b}\frac{\partial b}{\partial w}. \end{aligned} \tag{25.4.21}$$

この分解を使って $\frac{\partial f}{\partial w}$ を計算してみよう。ここで必要なのは、さまざまな 1 段階の偏微分だけだと分かる。

$$\begin{aligned} \frac{\partial f}{\partial u} &= 2(u+v), \quad \frac{\partial f}{\partial v} = 2(u+v), \\ \frac{\partial u}{\partial a} &= 2(a+b), \quad \frac{\partial u}{\partial b} = 2(a+b), \\ \frac{\partial v}{\partial a} &= 2(a-b), \quad \frac{\partial v}{\partial b} = -2(a-b), \\ \frac{\partial a}{\partial w} &= 2(w+x+y+z), \quad \frac{\partial b}{\partial w} = 2(w+x-y-z). \end{aligned} \tag{25.4.22}$$

これをコードに書くと、かなり扱いやすい式になる。

```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print(f'    f at {w}, {x}, {y}, {z} is {f}')

# Compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)

# Compute the final result from inputs to outputs
du_dw, dv_dw = du_da*da_dw + du_db*db_dw, dv_da*da_dw + dv_db*db_dw
df_dw = df_du*du_dw + df_dv*dv_dw
print(f'df/dw at {w}, {x}, {y}, {z} is {df_dw}')
```

```
    f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
```

しかし、これでも $\frac{\partial f}{\partial x}$ のようなものを簡単に計算できるわけではない。その理由は、連鎖律を どのように適用するかを選んだからである。上で行ったことを見ると、可能な限り常に分母に ∂w を残していた。このようにして、w が他のすべての変数をどう変えるかを見る形で連鎖律を適用したのである。もしそれが目的なら、それでよいだろう。しかし、深層学習での動機を思い出する。私たちは各パラメータが 損失をどう変えるかを見たいのである。本質的には、可能な限り常に分子に ∂f を残す形で連鎖律を適用したいのです！

より明示的に言うと、次のように書ける。

$$\begin{aligned}\frac{\partial f}{\partial w} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial w},\\ \frac{\partial f}{\partial a} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a},\\ \frac{\partial f}{\partial b} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial b} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial b}.\end{aligned} \tag{25.4.23}$$

この連鎖律の適用では、$\frac{\partial f}{\partial u}, \frac{\partial f}{\partial v}, \frac{\partial f}{\partial a}, \frac{\partial f}{\partial b}$, and $\frac{\partial f}{\partial w}$ を明示的に計算することになる。さらに次の式を加えても何の問題もない。

$$\begin{aligned}\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial x},\\ \frac{\partial f}{\partial y} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial y},\\ \frac{\partial f}{\partial z} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial z} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial z}.\end{aligned} \tag{25.4.24}$$

そして、ネットワーク全体のどのノードを変えたときにも f がどう変わるかを追跡できる。実装してみよう。

```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print(f'f at {w}, {x}, {y}, {z} is {f}')

# Compute the derivative using the decomposition above
# First compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)
da_dx, db_dx = 2*(w + x + y + z), 2*(w + x - y - z)
da_dy, db_dy = 2*(w + x + y + z), -2*(w + x - y - z)
da_dz, db_dz = 2*(w + x + y + z), -2*(w + x - y - z)

# Now compute how f changes when we change any value from output to input
df_da, df_db = df_du*du_da + df_dv*dv_da, df_du*du_db + df_dv*dv_db
df_dw, df_dx = df_da*da_dw + df_db*db_dw, df_da*da_dx + df_db*db_dx
df_dy, df_dz = df_da*da_dy + df_db*db_dy, df_da*da_dz + df_db*db_dz

print(f'df/dw at {w}, {x}, {y}, {z} is {df_dw}')
print(f'df/dx at {w}, {x}, {y}, {z} is {df_dx}')
print(f'df/dy at {w}, {x}, {y}, {z} is {df_dy}')
print(f'df/dz at {w}, {x}, {y}, {z} is {df_dz}')
```

```
f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
df/dx at -1, 0, -2, 1 is -4096
df/dy at -1, 0, -2, 1 is -4096
df/dz at -1, 0, -2, 1 is -4096
```

入力から出力へではなく、出力から入力へ向かって導関数を計算していること（上の最初のコード片で行ったように）が、このアルゴリズムに 逆伝播という名前を与えている。ここには 2 つの段階がある。1. 関数の値と 1 段階の偏微分を前から後ろへ計算する。上では分けて書いていないが、これは 1 つの 順伝播にまとめられる。2. f の勾配を後ろから前へ計算する。これを 逆伝播と呼ぶ。

これは、ネットワーク内のすべての重みに対する損失の勾配を 1 回の計算で求められるようにするために、あらゆる深層学習アルゴリズムが実装しているものそのものである。このような分解があるというのは驚くべき事実である。

これがどのようにまとめられるかを見るために、この例を少し見てみよう。

```
# Initialize as ndarrays, then attach gradients
w = torch.tensor([-1.], requires_grad=True)
x = torch.tensor([0.], requires_grad=True)
y = torch.tensor([-2.], requires_grad=True)
```

(continues on next page)

(continued from previous page)

```
z = torch.tensor([1.], requires_grad=True)
# Do the computation like usual, tracking gradients
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2

# Execute backward pass
f.backward()

print(f'df/dw at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {w.grad.data.item()}')
print(f'df/dx at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {x.grad.data.item()}')
print(f'df/dy at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {y.grad.data.item()}')
print(f'df/dz at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {z.grad.data.item()}')
```

```
df/dw at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dx at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dy at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dz at -1.0, 0.0, -2.0, 1.0 is -4096.0
```

```
# Initialize as ndarrays, then attach gradients
w, x, y, z = np.array(-1), np.array(0), np.array(-2), np.array(1)

w.attach_grad()
x.attach_grad()
y.attach_grad()
z.attach_grad()

# Do the computation like usual, tracking gradients
with autograd.record():
    a, b = (w + x + y + z)**2, (w + x - y - z)**2
    u, v = (a + b)**2, (a - b)**2
    f = (u + v)**2

# Execute backward pass
f.backward()

print(f'df/dw at {w}, {x}, {y}, {z} is {w.grad}')
print(f'df/dx at {w}, {x}, {y}, {z} is {x.grad}')
print(f'df/dy at {w}, {x}, {y}, {z} is {y.grad}')
print(f'df/dz at {w}, {x}, {y}, {z} is {z.grad}')
```

```
df/dw at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dx at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dy at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dz at -1.0, 0.0, -2.0, 1.0 is -4096.0
[07:20:46] ../src/base.cc:48: GPU context requested, but no GPUs found.
```

```
# Initialize as ndarrays, then attach gradients
w = tf.Variable(tf.constant([-1.]))
x = tf.Variable(tf.constant([0.]))
y = tf.Variable(tf.constant([-2.]))
z = tf.Variable(tf.constant([1.]))
# Do the computation like usual, tracking gradients
with tf.GradientTape(persistent=True) as t:
    a, b = (w + x + y + z)**2, (w + x - y - z)**2
    u, v = (a + b)**2, (a - b)**2
    f = (u + v)**2

# Execute backward pass
w_grad = t.gradient(f, w).numpy()
x_grad = t.gradient(f, x).numpy()
y_grad = t.gradient(f, y).numpy()
z_grad = t.gradient(f, z).numpy()

print(f'df/dw at {w.numpy()}, {x.numpy()}, {y.numpy()}, '
      f'{z.numpy()} is {w_grad}')
print(f'df/dx at {w.numpy()}, {x.numpy()}, {y.numpy()}, '
      f'{z.numpy()} is {x_grad}')
print(f'df/dy at {w.numpy()}, {x.numpy()}, {y.numpy()}, '
      f'{z.numpy()} is {y_grad}')
print(f'df/dz at {w.numpy()}, {x.numpy()}, {y.numpy()}, '
      f'{z.numpy()} is {z_grad}')
```

```
df/dw at [-1.], [0.], [-2.], [1.] is [-4096.]
df/dx at [-1.], [0.], [-2.], [1.] is [-4096.]
df/dy at [-1.], [0.], [-2.], [1.] is [-4096.]
df/dz at [-1.], [0.], [-2.], [1.] is [-4096.]
```

上で行ったことはすべて、`f.backwards()` を呼び出すだけで自動的に実行できる。

25.4.6 ヘッシアン

単変数微積分と同様に、勾配だけを使うよりも関数をよりよく近似するために、高階導関数を考えることは有用である。

複数変数の関数の高階導関数を扱うとき、すぐに直面する問題が 1 つある。それは、その数が非常に多いことである。n 変数の関数 $f(x_1, \ldots, x_n)$ があるとき、2 階導関数は n^2 個取れる。すなわち、i と j の任意の組に対して

$$\frac{d^2 f}{dx_i dx_j} = \frac{d}{dx_i}\left(\frac{d}{dx_j} f\right). \tag{25.4.25}$$

これは伝統的に ヘッシアンと呼ばれる行列にまとめられる。

$$\mathbf{H}_f = \begin{bmatrix} \frac{d^2 f}{dx_1 dx_1} & \cdots & \frac{d^2 f}{dx_1 dx_n} \\ \vdots & \ddots & \vdots \\ \frac{d^2 f}{dx_n dx_1} & \cdots & \frac{d^2 f}{dx_n dx_n} \end{bmatrix}. \tag{25.4.26}$$

この行列の各要素が独立というわけではない。実際、混合偏導関数（複数の変数に関する偏導関数）が存在し連続である限り、任意の i と j について

$$\frac{d^2 f}{dx_i dx_j} = \frac{d^2 f}{dx_j dx_i}. \tag{25.4.27}$$

これは、まず x_i の方向に関数を摂動し、その後 x_j で摂動した結果と、先に x_j で摂動し、その後 x_i で摂動した結果を比較することで従う。どちらの順序でも、f の出力に対する最終的な変化は同じになるからである。

単変数の場合と同様に、これらの導関数を使うと、ある点の近くで関数がどう振る舞うかをはるかによく把握できる。特に、単変数で見たように、点 $\mathbf{x}_0$ の近くで最もよく当てはまる 2 次式を見つけるのに使える。

例を見てみよう。$f(x_1, x_2) = a + b_1 x_1 + b_2 x_2 + c_{11} x_1^2 + c_{12} x_1 x_2 + c_{22} x_2^2$ とする。これは 2 変数の 2 次式の一般形である。関数値、その勾配、そして (25.4.26) のヘッシアンを、すべて原点で見てみると、

$$\begin{aligned} f(0,0) &= a, \\ \nabla f(0,0) &= \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \\ \mathbf{H} f(0,0) &= \begin{bmatrix} 2c_{11} & c_{12} \\ c_{12} & 2c_{22} \end{bmatrix}, \end{aligned} \tag{25.4.28}$$

元の多項式は次のようにして復元できる。

$$f(\mathbf{x}) = f(0) + \nabla f(0) \cdot \mathbf{x} + \frac{1}{2} \mathbf{x}^\top \mathbf{H} f(0) \mathbf{x}. \tag{25.4.29}$$

一般に、任意の点 $\mathbf{x}_0$ でこの展開を計算すると、

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^\top \mathbf{H} f(\mathbf{x}_0) (\mathbf{x} - \mathbf{x}_0). \tag{25.4.30}$$

これは任意次元の入力に対して成り立ち、ある点での関数の最良の 2 次近似を与える。例として、次の関数をプロットしてみよう。

$$f(x, y) = x e^{-x^2 - y^2}. \tag{25.4.31}$$

勾配とヘッシアンは

$$\nabla f(x, y) = e^{-x^2 - y^2} \begin{pmatrix} 1 - 2x^2 \\ -2xy \end{pmatrix} \text{ and } \mathbf{H} f(x, y) = e^{-x^2 - y^2} \begin{pmatrix} 4x^3 - 6x & 4x^2 y - 2y \\ 4x^2 y - 2y & 4xy^2 - 2x \end{pmatrix}. \tag{25.4.32}$$

したがって、少し計算すると、$[-1, 0]^\top$ における近似 2 次式は

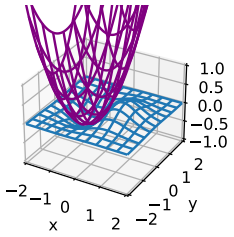
$$f(x, y) \approx e^{-1} \left(-1 - (x + 1) + (x + 1)^2 + y^2 \right). \tag{25.4.33}$$

```
# Construct grid and compute function
x, y = torch.meshgrid(torch.linspace(-2, 2, 101),
                      torch.linspace(-2, 2, 101))

z = x*torch.exp(- x**2 - y**2)

# Compute approximating quadratic with gradient and Hessian at (1, 0)
w = torch.exp(torch.tensor([-1.]))*(-1 - (x + 1) + 2 * (x + 1)**2 + 2 * y**2)

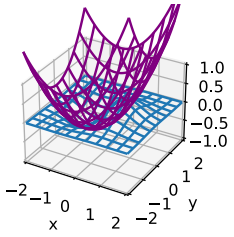
# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x.numpy(), y.numpy(), z.numpy(),
                  **{'rstride': 10, 'cstride': 10})
ax.plot_wireframe(x.numpy(), y.numpy(), w.numpy(),
                  **{'rstride': 10, 'cstride': 10}, color='purple')
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(-1, 1)
ax.dist = 12
```




```
# Construct grid and compute function
x, y = np.meshgrid(np.linspace(-2, 2, 101),
                   np.linspace(-2, 2, 101), indexing='ij')
z = x*np.exp(- x**2 - y**2)

# Compute approximating quadratic with gradient and Hessian at (1, 0)
w = np.exp(-1)*(-1 - (x + 1) + (x + 1)**2 + y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x.asnumpy(), y.asnumpy(), z.asnumpy(),
                  **{'rstride': 10, 'cstride': 10})
ax.plot_wireframe(x.asnumpy(), y.asnumpy(), w.asnumpy(),
                  **{'rstride': 10, 'cstride': 10}, color='purple')
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(-1, 1)
ax.dist = 12
```

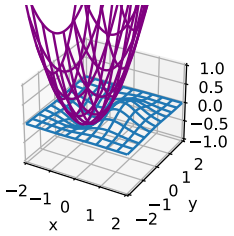



```
# Construct grid and compute function
x, y = tf.meshgrid(tf.linspace(-2., 2., 101),
                   tf.linspace(-2., 2., 101))

z = x*tf.exp(- x**2 - y**2)

# Compute approximating quadratic with gradient and Hessian at (1, 0)
w = tf.exp(tf.constant([-1.]))*(-1 - (x + 1) + 2 * (x + 1)**2 + 2 * y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x.numpy(), y.numpy(), z.numpy(),
                  **{'rstride': 10, 'cstride': 10})
ax.plot_wireframe(x.numpy(), y.numpy(), w.numpy(),
                  **{'rstride': 10, 'cstride': 10}, color='purple')
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(-1, 1)
ax.dist = 12
```



これは Section 15.3 で議論するニュートン法の基礎を成する。そこでは、最もよく当てはまる 2 次式を反復的に求め、その 2 次式を正確に最小化する。

25.4.7 少しの行列微積分

行列を含む関数の導関数は、実は特に扱いやすいことが分かる。この節は記法が重くなるので、最初の読みでは飛ばしても構わないが、行列を含む関数の導関数が、特に深層学習で中心的な役割を果たす行列演算を考えると、最初に思うよりずっとすっきりしていることを知っておくのは有益である。

まず例から始めよう。固定された列ベクトル $\boldsymbol{\beta}$ があり、積の関数 $f(\mathbf{x}) = \boldsymbol{\beta}^\top \mathbf{x}$ を考えて、$\mathbf{x}$ を変えたときに内積がどう変わるかを理解したいとする。

機械学習で行列の導関数を扱うときに役立つ記法として、分母配置の行列微分と呼ばれるものがある。これは、微分の分母にあるベクトル、行列、あるいはテンソルの形に合わせて偏微分を並べるものである。この場合、次のように書く。

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix}, \tag{25.4.34}$$

ここでは列ベクトル $\mathbf{x}$ の形に合わせている。

関数を成分ごとに書くと、

$$f(\mathbf{x}) = \sum_{i=1}^{n} \beta_i x_i = \beta_1 x_1 + \cdots + \beta_n x_n. \tag{25.4.35}$$

ここで、たとえば β_1 に関する偏微分をとると、最初の項だけが β_1 に x_1 を掛けたものなので、それ以外はすべてゼロである。したがって

$$\frac{df}{dx_1} = \beta_1, \tag{25.4.36}$$

より一般には

$$\frac{df}{dx_i} = \beta_i. \tag{25.4.37}$$

これを行列としてまとめ直すと、

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} = \boldsymbol{\beta}. \tag{25.4.38}$$

これは、この節で何度も出てくる行列微積分のいくつかの特徴を示している。

- まず、計算はかなり複雑になる。
- 次に、最終結果は途中の過程よりずっとすっきりしており、常に単変数の場合と似た形になる。この場合、$\frac{d}{dx}(bx) = b$ と $\frac{d}{d\mathbf{x}}(\boldsymbol{\beta}^\top \mathbf{x}) = \boldsymbol{\beta}$ が似ていることに注意する。
- 第 3 に、転置がどこからともなく現れることがよくある。その根本的な理由は、分母の形に合わせるという約束にある。したがって行列を掛け合わせるときには、元の項の形に戻すために転置を取る必要があるのである。

直感をさらに深めるために、もう少し難しい計算をしてみよう。列ベクトル $\mathbf{x}$ と正方行列 A があり、次を計算したいとする。

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A \mathbf{x}). \tag{25.4.39}$$

扱いやすい記法にするため、アインシュタイン記法を使ってこの問題を考えよう。この場合、関数は

$$\mathbf{x}^\top A\mathbf{x} = x_i a_{ij} x_j. \tag{25.4.40}$$

と書ける。

導関数を計算するには、各 k について

$$\frac{d}{dx_k}(\mathbf{x}^\top A\mathbf{x}) = \frac{d}{dx_k} x_i a_{ij} x_j. \tag{25.4.41}$$

の値を理解する必要がある。

積の法則より、これは

$$\frac{d}{dx_k} x_i a_{ij} x_j = \frac{dx_i}{dx_k} a_{ij} x_j + x_i a_{ij} \frac{dx_j}{dx_k}. \tag{25.4.42}$$

となる。$\frac{dx_i}{dx_k}$ のような項は、$i = k$ のとき 1、それ以外では 0 であることは難しくない。つまり、i と k が異なる項はこの和から消え、最初の和に残るのは $i = k$ のものだけである。同じ理屈が第 2 項にも当てはまり、そこでは $j = k$ が必要である。これにより

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{kj} x_j + x_i a_{ik}. \tag{25.4.43}$$

となる。

ここで、アインシュタイン記法における添字の名前は任意である。i と j が異なるという事実は、この計算では本質的ではないので、添字を付け替えて両方とも i を使うようにすると、

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{ki} x_i + x_i a_{ik} = (a_{ki} + a_{ik}) x_i. \tag{25.4.44}$$

となる。

ここから先に進むには少し練習が必要である。この結果を行列演算の形で表してみよう。$a_{ki} + a_{ik}$ は $\mathbf{A} + \mathbf{A}^\top$ の k, i 成分である。したがって

$$\frac{d}{dx_k} x_i a_{ij} x_j = [\mathbf{A} + \mathbf{A}^\top]_{ki} x_i. \tag{25.4.45}$$

となる。

同様に、この項は行列 $\mathbf{A} + \mathbf{A}^\top$ とベクトル $\mathbf{x}$ の積なので、

$$\left[\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x})\right]_k = \frac{d}{dx_k} x_i a_{ij} x_j = [(\mathbf{A} + \mathbf{A}^\top)\mathbf{x}]_k. \tag{25.4.46}$$

と分かる。

したがって、(25.4.39) で求めた導関数の k 番目の成分は、右辺ベクトルの k 番目の成分と同じであり、両者は等しいことが分かる。よって

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}. \tag{25.4.47}$$

これは前の例よりかなり多くの作業を要したが、最終結果は短いものである。さらに、通常の単変数微分について次を考えてみよ。

$$\frac{d}{dx}(xax) = \frac{dx}{dx}ax + xa\frac{dx}{dx} = (a+a)x. \tag{25.4.48}$$

同様に $\frac{d}{dx}(ax^2) = 2ax = (a+a)x$ である。やはり、単変数の場合に似た結果が得られるが、そこに転置が 1 つ加わっている。

ここまで来ると、このパターンはかなり怪しく見えてくるはずである。なぜそうなるのか考えてみよう。このような行列微分をするとき、まず得られる式は別の行列式だと仮定してみる。つまり、行列とその転置の積や和で書ける式である。そのような式が存在するなら、すべての行列に対して成り立たなければならない。特に、1×1 行列に対しても成り立つ必要がある。その場合、行列積は単なる数の積、行列和は単なる和であり、転置は何の役にも立たない！つまり、どんな式であれ、必ず単変数の式と一致しなければならないのである。したがって、少し練習すれば、対応する単変数の式がどうなるべきかを知っているだけで、行列の導関数をしばしば推測できるのである！

これを試してみよう。$\mathbf{X}$ が $n \times m$ 行列、$\mathbf{U}$ が $n \times r$、$\mathbf{V}$ が $r \times m$ だとする。次を計算してみよう。

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = ? \tag{25.4.49}$$

この計算は行列分解と呼ばれる分野で重要である。しかしここでは、単に計算すべき導関数にすぎない。これが 1×1 行列ならどうなるか想像してみよう。その場合、次の式になる。

$$\frac{d}{dv}(x - uv)^2 = -2(x - uv)u, \tag{25.4.50}$$

これはごく標準的な導関数である。これを行列式に戻そうとすると、

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2(\mathbf{X} - \mathbf{U}\mathbf{V})\mathbf{U}. \tag{25.4.51}$$

となる。

しかし、これを見ると少しおかしいことに気づく。$\mathbf{X}$ は $n \times m$ であり、$\mathbf{U}\mathbf{V}$ も同じなので、行列 $2(\mathbf{X} - \mathbf{U}\mathbf{V})$ は $n \times m$ である。一方、$\mathbf{U}$ は $n \times r$ であり、$n \times m$ 行列と $n \times r$ 行列は次元が合わないので掛けられない！

私たちが求めたいのは $\frac{d}{d\mathbf{V}}$ であり、これは $\mathbf{V}$ と同じ形、つまり $r \times m$ である。したがって、どうにかして $n \times m$ 行列と $n \times r$ 行列を掛け合わせて（おそらく転置を使って）$r \times m$ を得る必要がある。これは $U^\top$ を $(\mathbf{X} - \mathbf{U}\mathbf{V})$ に掛けることで実現できる。したがって、(25.4.49) の解は

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2\mathbf{U}^\top(\mathbf{X} - \mathbf{U}\mathbf{V}). \tag{25.4.52}$$

だと推測できる。

これが正しいことを示すには、詳細な計算を省くわけにはいきない。この経験

則が正しいとすでに信じているなら、この導出は飛ばして構いない。次を計算するには

$$\frac{d}{d\mathbf{V}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2, \tag{25.4.53}$$

各 a と b について

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = \frac{d}{dv_{ab}}\sum_{i,j}\left(x_{ij}-\sum_k u_{ik}v_{kj}\right)^2. \tag{25.4.54}$$

を求めなければならない。

$\frac{d}{dv_{ab}}$ に関しては $\mathbf{X}$ と $\mathbf{U}$ のすべての要素が定数であることを思い出すと、微分を和の中に押し込み、2 乗に連鎖律を適用して

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = \sum_{i,j} 2\left(x_{ij}-\sum_k u_{ik}v_{kj}\right)\left(-\sum_k u_{ik}\frac{dv_{kj}}{dv_{ab}}\right). \tag{25.4.55}$$

となる。

前の導出と同様に、$\frac{dv_{kj}}{dv_{ab}}$ は $k=a$ かつ $j=b$ のときにのみ非ゼロであることに注意できる。どちらか一方でも満たさなければ、その項はゼロなので自由に捨てられる。すると

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = -2\sum_i\left(x_{ib}-\sum_k u_{ik}v_{kb}\right)u_{ia}. \tag{25.4.56}$$

となる。

ここで重要な微妙さは、$k=a$ という条件は内側の和の中には現れないということである。なぜなら、その k は内側の項の中で和をとっているダミー変数だからである。記法をよりきれいにした例として、なぜ

$$\frac{d}{dx_1}\left(\sum_i x_i\right)^2 = 2\left(\sum_i x_i\right) \tag{25.4.57}$$

となるのかを考えてみよ。

ここから、和の成分を識別し始められる。まず、

$$\sum_k u_{ik}v_{kb} = [\mathbf{U}\mathbf{V}]_{ib}. \tag{25.4.58}$$

したがって、和の中の内側の式全体は

$$x_{ib}-\sum_k u_{ik}v_{kb} = [\mathbf{X}-\mathbf{U}\mathbf{V}]_{ib}. \tag{25.4.59}$$

となる。

これにより、導関数を次のように書ける。

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = -2\sum_i [\mathbf{X}-\mathbf{U}\mathbf{V}]_{ib}u_{ia}. \tag{25.4.60}$$

これを行列の a, b 成分の形にしたいので、前の例と同じ技法を使って行列式にまとめるには、u_{ia} の添字の順序を入れ替える必要がある。$u_{ia} = [\mathbf{U}^\top]_{ai}$ に気づけば、

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2\sum_i [\mathbf{U}^\top]_{ai}[\mathbf{X} - \mathbf{UV}]_{ib}. \tag{25.4.61}$$

と書ける。

これは行列積なので、次のように結論できる。

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2[\mathbf{U}^\top(\mathbf{X} - \mathbf{UV})]_{ab}. \tag{25.4.62}$$

したがって、(25.4.49) の解は

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2\mathbf{U}^\top(\mathbf{X} - \mathbf{UV}). \tag{25.4.63}$$

と書ける。

これは上で推測した解と一致している！

ここで、「これまで学んだ微積分の規則をすべてそのまま行列版で書けないのだろうか？まだ機械的なことは明らかなのだから、さっさと済ませればよいのでは？」と思うのはもっともである。実際、そのような規則はあり、(Petersen and Pedersen, 2008) に優れた要約がある。しかし、単一の値に比べて行列演算の組み合わせ方が非常に多いため、行列の導関数の規則は単変数の場合よりはるかに多くなる。多くの場合、添字を使って扱うか、適切な場合には自動微分に任せるのが最善である。

25.4.8 まとめ

- 高次元では、1 次元の導関数と同じ役割を果たす勾配を定義できる。これにより、入力に任意の小さな変化を与えたときに多変数関数がどう変わるかを調べられる。
- 逆伝播アルゴリズムは、多変数の連鎖律を整理して、多くの偏微分を効率よく計算する方法とみなせる。
- 行列微積分を使うと、行列式の導関数を簡潔に書ける。

25.4.9 演習

1. 列ベクトル $\boldsymbol{\beta}$ が与えられたとき、$f(\mathbf{x}) = \boldsymbol{\beta}^\top\mathbf{x}$ と $g(\mathbf{x}) = \mathbf{x}^\top\boldsymbol{\beta}$ の両方の導関数を求めよ。なぜ同じ答えになるのか。
2. $\mathbf{v}$ を n 次元ベクトルとする。$\frac{\partial}{\partial\mathbf{v}}\|\mathbf{v}\|_2$ は何か。
3. $L(x, y) = \log(e^x + e^y)$ とする。勾配を求めよ。勾配の成分の和はいくつか。
4. $f(x, y) = x^2y + xy^2$ とする。臨界点が $(0, 0)$ のみであることを示せ。また、$f(x, x)$ を考えることで、$(0, 0)$ が最大値か最小値か、それともどちらでもないかを判定せよ。

5. $f(\mathbf{x}) = g(\mathbf{x}) + h(\mathbf{x})$ を最小化しているとする。$\nabla f = 0$ の条件を g と h の観点から幾何学的にどう解釈できるか。

25.5 積分計算

微分は、伝統的な微積分教育の内容の半分にすぎない。もう一つの柱である積分は、最初は「この曲線の下の面積はどれくらいか？」という、かなり別個の問いのように見える。一見無関係に思えるが、積分は 微積分の基本定理として知られるものを通じて、微分と密接に結びついている。

本書で扱う機械学習のレベルでは、積分について深い理解は必要ない。しかし、後で出てくる応用の土台を築くために、簡単な導入を行いる。

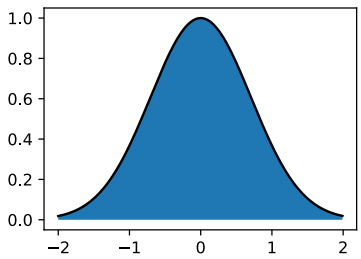
25.5.1 幾何学的解釈

関数 $f(x)$ があるとする。簡単のため、$f(x)$ は非負（0 未満の値を取らない）と仮定しよう。ここで理解したいのは、$f(x)$ と x 軸の間に含まれる面積がどれくらいか、ということである。

```
%matplotlib inline
from d2l import torch as d2l
from IPython import display
from mpl_toolkits import mplot3d
import torch

x = torch.arange(-2, 2, 0.01)
f = torch.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist(), f.tolist())
d2l.plt.show()
```




```
%matplotlib inline
from d2l import mxnet as d2l
```

(continues on next page)

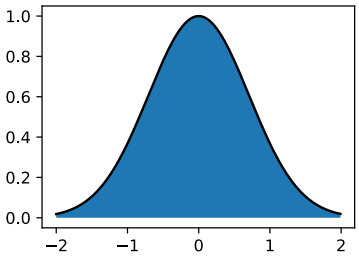
(continued from previous page)

```
from IPython import display
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()

x = np.arange(-2, 2, 0.01)
f = np.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist(), f.tolist())
d2l.plt.show()
```

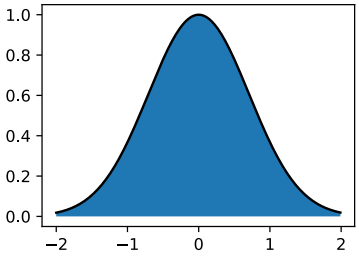
```
[07:00:00] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```




```
%matplotlib inline
from d2l import tensorflow as d2l
from IPython import display
from mpl_toolkits import mplot3d
import tensorflow as tf

x = tf.range(-2, 2, 0.01)
f = tf.exp(-x**2)

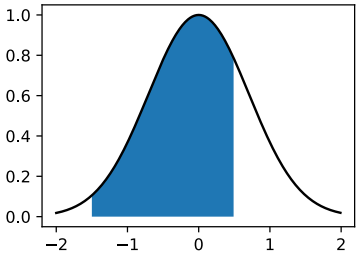
d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.numpy(), f.numpy())
d2l.plt.show()
```



多くの場合、この面積は無限大になるか、あるいは定義できない（$f(x) = x^2$ の下の面積を考えてみてほしい）。そのため、通常は a と b という 2 つの端点の間の面積を考える。

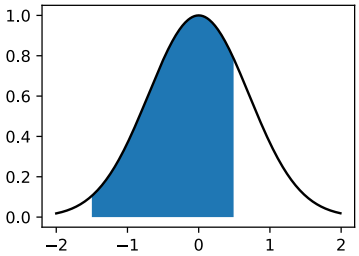
```
x = torch.arange(-2, 2, 0.01)
f = torch.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist()[50:250], f.tolist()[50:250])
d2l.plt.show()
```



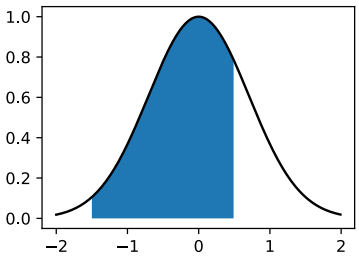

```
x = np.arange(-2, 2, 0.01)
f = np.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist()[50:250], f.tolist()[50:250])
d2l.plt.show()
```




```
x = tf.range(-2, 2, 0.01)
f = tf.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.numpy()[50:250], f.numpy()[50:250])
d2l.plt.show()
```



この面積は、以下の積分記号で表する。

$$\text{Area}(\mathcal{A}) = \int_a^b f(x)\,dx. \tag{25.5.1}$$

内部の変数はダミー変数であり、$\sum$ における和の添字と同じようなものである。そのため、内部の変数は任意の文字で書き換えても同じ意味になる。

$$\int_a^b f(x)\,dx = \int_a^b f(z)\,dz. \tag{25.5.2}$$

このような積分を近似する方法には、昔からの考え方がある。区間 a から b の間の領域を N 個の縦のスライスに分割することを考える。N が十分大きければ、各スライスの面積を長方形で近似し、それらを足し合わせることで曲線の下の総面積を求められる。これをコードで行う例を見てみよう。真の値の求め方は後の節で説明する。

```
epsilon = 0.05
a = 0
b = 2

x = torch.arange(a, b, epsilon)
f = x / (1 + x**2)

approx = torch.sum(epsilon*f)
true = torch.log(torch.tensor([5.])) / 2

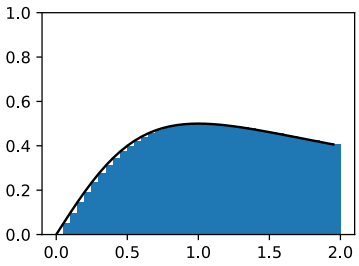
d2l.set_figsize()
d2l.plt.bar(x, f, width=epsilon, align='edge')
d2l.plt.plot(x, f, color='black')
d2l.plt.ylim([0, 1])
d2l.plt.show()

f'approximation: {approx}, truth: {true}'
```

```
'approximation: 0.7944855690002441, truth: tensor([0.8047])'
```

```
epsilon = 0.05
a = 0
b = 2
```

(continues on next page)



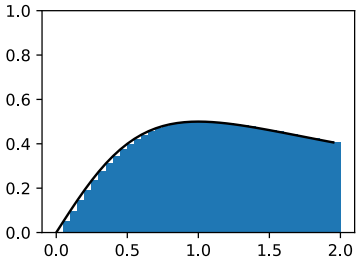
(continued from previous page)

```
x = np.arange(a, b, epsilon)
f = x / (1 + x**2)

approx = np.sum(epsilon*f)
true = np.log(2) / 2

d2l.set_figsize()
d2l.plt.bar(x.asnumpy(), f.asnumpy(), width=epsilon, align='edge')
d2l.plt.plot(x, f, color='black')
d2l.plt.ylim([0, 1])
d2l.plt.show()

f'approximation: {approx}, truth: {true}'
```




```
'approximation: 0.7944855690002441, truth: 0.34657359027997264'
```

```
epsilon = 0.05
a = 0
b = 2

x = tf.range(a, b, epsilon)
f = x / (1 + x**2)

approx = tf.reduce_sum(epsilon*f)
true = tf.math.log(tf.constant([5.])) / 2

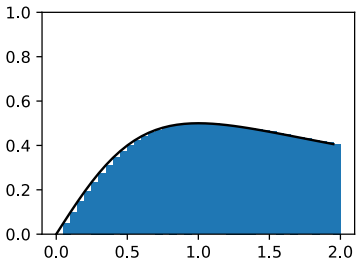
d2l.set_figsize()
```

(continues on next page)

(continued from previous page)

```
d2l.plt.bar(x, f, width=epsilon, align='edge')
d2l.plt.plot(x, f, color='black')
d2l.plt.ylim([0, 1])
d2l.plt.show()

f'approximation: {approx}, truth: {true}'
```




```
'approximation: 0.7944855690002441, truth: [0.804719]'
```

問題は、数値的にはできるとしても、この方法を解析的に扱えるのは次のような最も単純な関数に限られることである。

$$\int_a^b x \, dx. \tag{25.5.3}$$

コード中の例のような、もう少し複雑なもの

$$\int_a^b \frac{x}{1+x^2} \, dx. \tag{25.5.4}$$

は、このような直接的な方法では解けない。

そこで別のアプローチを取りる。面積という概念を直感的に扱いながら、積分を求めるための主要な計算道具である 微積分の基本定理を学ぶ。これが積分の学習の基礎になる。

25.5.2 微積分の基本定理

積分の理論をさらに深く見ていくために、次の関数を導入しよう。

$$F(x) = \int_0^x f(y)dy. \tag{25.5.5}$$

この関数は、x をどのように変えるかに応じて、0 から x までの面積を表する。次が成り立つので、これで十分である。

$$\int_a^b f(x) \, dx = F(b) - F(a). \tag{25.5.6}$$

これは、図 Fig. 25.5.1 に示すように、遠い端点までの面積を求めてから、近い端点までの面積を引けばよい、という事実を数学的に表したものである。

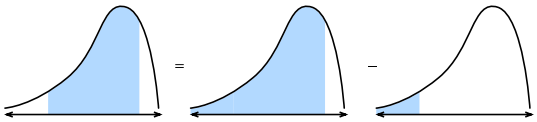


Fig. 25.5.1 曲線の下の面積を 2 点間で求める問題を、ある点の左側の面積を求める問題に帰着できる理由の可視化。

したがって、$F(x)$ が何であるかを求めれば、任意の区間での積分を求められる。

そのために、実験を考えてみよう。微積分でよく行うように、値をほんの少しだけずらしたときに何が起こるかを想像する。上のコメントから、

$$F(x+\epsilon) - F(x) = \int_x^{x+\epsilon} f(y)\, dy. \tag{25.5.7}$$

であることがわかる。これは、関数のごく薄い一片の下の面積だけ、関数値が変化することを意味する。

ここで近似を行いる。このような小さな面積の一片を見ると、その面積は、高さが $f(x)$、底辺の幅が ϵ の長方形の面積に近いように見える。実際、$\epsilon \to 0$ とすると、この近似はますます良くなることが示せる。したがって、

$$F(x+\epsilon) - F(x) \approx \epsilon f(x). \tag{25.5.8}$$

しかし、これは F の導関数を計算しているときに期待される形とまったく同じである。したがって、次の少し驚くべき事実がわかる。

$$\frac{dF}{dx}(x) = f(x). \tag{25.5.9}$$

これが 微積分の基本定理である。展開して書くと、

$$\frac{d}{dx}\int_0^x f(y)\, dy = f(x). \tag{25.5.10}$$

これは、面積を求めるという概念（*a priori* にはかなり難しい）を、導関数に関する主張（はるかによく理解されているもの）へと還元する。最後にもう 1 つ述べておくべきことは、これだけでは $F(x)$ が正確に何であるかはわからないという点である。実際、任意の C に対して $F(x)+C$ も同じ導関数を持つ。これは積分理論における避けられない事実である。幸いなことに、定積分を扱うときには定数項は打ち消し合うので、結果には影響しない。

$$\int_a^b f(x)\, dx = (F(b)+C) - (F(a)+C) = F(b) - F(a). \tag{25.5.11}$$

これは抽象的なナンセンスのように見えるかもしれないが、積分の計算に対してまったく新しい見方を与えてくれることに少し立ち止まって注目しよう。もはや、面積を復元するために何かを切り分けて足し合わせる必要はない。必要なのは、求めたい関数の導関数が与えられた関数になるような関数を見つけることだけである。これは非常に強力である。なぜなら、Section 25.3.2 の表を逆向きに見るだけで、多くの難しい積分を書き下せるからである。たとえば、x^n

の導関数は nx^{n-1} であることを知っている。したがって、基本定理 (25.5.10) を用いると、

$$\int_0^x ny^{n-1}\,dy = x^n - 0^n = x^n. \tag{25.5.12}$$

同様に、e^x の導関数はそれ自身だろうから、

$$\int_0^x e^x\,dx = e^x - e^0 = e^x - 1. \tag{25.5.13}$$

このようにして、微分積分学の考え方を自由に活用しながら、積分の理論全体を構築できる。すべての積分法則は、この 1 つの事実から導かれる。

25.5.3 変数変換

微分と同様に、積分の計算を扱いやすくするいくつかの規則がある。実際、微分積分学の各規則（積の法則、和の法則、連鎖律など）には、それぞれ対応する積分計算の規則（部分積分、積分の線形性、変数変換公式）がある。この節では、その中でもおそらく最も重要なもの、変数変換公式について見ていきる。

まず、積分そのものである関数を考える。

$$F(x) = \int_0^x f(y)\,dy. \tag{25.5.14}$$

この関数を別の関数と合成して $F(u(x))$ を得たとき、これがどのような形になるかを知りたいとする。連鎖律より、

$$\frac{d}{dx}F(u(x)) = \frac{dF}{du}(u(x)) \cdot \frac{du}{dx}. \tag{25.5.15}$$

これを、上と同様に基本定理 (25.5.10) を用いて積分に関する主張へ変換できる。すると、

$$F(u(x)) - F(u(0)) = \int_0^x \frac{dF}{du}(u(y)) \cdot \frac{du}{dy}\,dy. \tag{25.5.16}$$

ここで、F が積分そのものであることを思い出すと、左辺は次のように書き換えられる。

$$\int_{u(0)}^{u(x)} f(y)\,dy = \int_0^x \frac{dF}{du}(u(y)) \cdot \frac{du}{dy}\,dy. \tag{25.5.17}$$

同様に、F が積分であることから、基本定理 (25.5.10) を用いて $\frac{dF}{dx} = f$ と認識できるので、結局

$$\int_{u(0)}^{u(x)} f(y)\,dy = \int_0^x f(u(y)) \cdot \frac{du}{dy}\,dy. \tag{25.5.18}$$

これが 変数変換の公式である。

より直感的に導くために、$f(u(x))$ の積分を x と $x+\epsilon$ の間で考えてみよう。ϵ が小さいとき、この積分は対応する長方形の面積である $\epsilon f(u(x))$ に近似され

る。次に、$f(y)$ を $u(x)$ から $u(x+\epsilon)$ まで積分したものと比較する。$u(x+\epsilon) \approx u(x) + \epsilon \frac{du}{dx}(x)$ であることから、この長方形の面積はおよそ $\epsilon \frac{du}{dx}(x) f(u(x))$ である。したがって、これら 2 つの長方形の面積を一致させるには、図 Fig. 25.5.2 に示すように、最初のものに $\frac{du}{dx}(x)$ を掛ける必要がある。

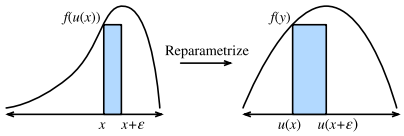


Fig. 25.5.2 変数変換の下で 1 つの細い長方形がどのように変換されるかの可視化。

これは、

$$\int_x^{x+\epsilon} f(u(y)) \frac{du}{dy}(y)\, dy = \int_{u(x)}^{u(x+\epsilon)} f(y)\, dy. \tag{25.5.19}$$

が成り立つことを意味する。

これは、1 つの小さな長方形について表した変数変換公式である。

$u(x)$ と $f(x)$ を適切に選べば、非常に複雑な積分の計算が可能になる。たとえば、$f(y) = 1$、$u(x) = e^{-x^2}$ を選ぶと（したがって $\frac{du}{dx}(x) = -2xe^{-x^2}$）、例えば

$$e^{-1} - 1 = \int_{e^{-0}}^{e^{-1}} 1\, dy = -2 \int_0^1 y e^{-y^2}\, dy, \tag{25.5.20}$$

が示せる。したがって、整理すると

$$\int_0^1 y e^{-y^2}\, dy = \frac{1 - e^{-1}}{2}. \tag{25.5.21}$$

25.5.4 符号規約に関する注意

注意深い読者は、上の計算に何か奇妙な点があることに気づくだろう。たとえば、

$$\int_{e^{-0}}^{e^{-1}} 1\, dy = e^{-1} - 1 < 0, \tag{25.5.22}$$

のような計算は負の数を生みる。面積について考えると、負の値を見るのは奇妙に感じられるかもしれない。そのため、ここでの規約を確認しておく価値がある。

数学者は、符号付き面積という概念を採用する。これは 2 つの形で現れる。まず、関数 $f(x)$ がときどき 0 未満になる場合、その面積も負になる。たとえば、

$$\int_0^1 (-1)\, dx = -1. \tag{25.5.23}$$

同様に、左から右ではなく右から左へ進む積分も、負の面積とみなされる。

$$\int_{0}^{-1} 1 \, dx = -1. \tag{25.5.24}$$

標準的な面積（正の関数を左から右へ積分したもの）は常に正である。これを反転させたもの、たとえば x 軸について反転して負の数の積分にしたり、y 軸について反転して順序が逆の積分にしたりすると、負の面積になる。そして実際、2 回反転すると 2 つの負号が打ち消し合って正の面積になる。

$$\int_{0}^{-1} (-1) \, dx = 1. \tag{25.5.25}$$

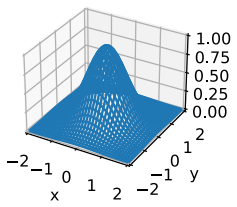
この話に聞き覚えがあるなら、その通りである。Section 25.1 では、行列式が同じように符号付き面積を表すことを説明した。

25.5.5 多重積分

場合によっては、より高次元で考える必要がある。たとえば、2 変数の関数 $f(x, y)$ があり、x が $[a, b]$ を、y が $[c, d]$ を動くときの f の下の体積を知りたいとする。

```
# Construct grid and compute function
x, y = torch.meshgrid(torch.linspace(-2, 2, 101), torch.linspace(-2, 2, 101))
z = torch.exp(- x**2 - y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.plt.xticks([-2, -1, 0, 1, 2])
d2l.plt.yticks([-2, -1, 0, 1, 2])
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(0, 1)
ax.dist = 12
```



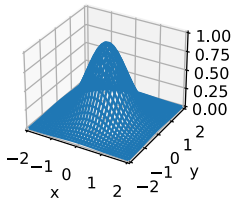

```
# Construct grid and compute function
x, y = np.meshgrid(np.linspace(-2, 2, 101), np.linspace(-2, 2, 101),
```

(continues on next page)

(continued from previous page)

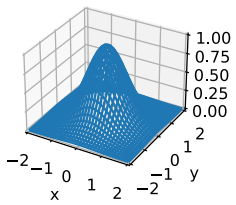
```
                    indexing='ij')
z = np.exp(- x**2 - y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x.asnumpy(), y.asnumpy(), z.asnumpy())
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.plt.xticks([-2, -1, 0, 1, 2])
d2l.plt.yticks([-2, -1, 0, 1, 2])
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(0, 1)
ax.dist = 12
```




```
# Construct grid and compute function
x, y = tf.meshgrid(tf.linspace(-2., 2., 101), tf.linspace(-2., 2., 101))
z = tf.exp(- x**2 - y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.plt.xticks([-2, -1, 0, 1, 2])
d2l.plt.yticks([-2, -1, 0, 1, 2])
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(0, 1)
ax.dist = 12
```



これを次のように書く。

$$\int_{[a,b]\times[c,d]} f(x,y)\,dx\,dy. \tag{25.5.26}$$

この積分を計算したいとする。私の主張は、まず x について積分し、その後で y について積分するという手順を繰り返すことで計算できる、ということである。すなわち、

$$\int_{[a,b]\times[c,d]} f(x,y)\,dx\,dy = \int_c^d \left(\int_a^b f(x,y)\,dx\right) dy. \tag{25.5.27}$$

なぜそうなるのか見てみよう。

上の図では、関数を $\epsilon \times \epsilon$ の正方形に分割し、それぞれを整数座標 i, j で表している。この場合、積分はおよそ

$$\sum_{i,j} \epsilon^2 f(\epsilon i, \epsilon j). \tag{25.5.28}$$

と近似できる。

一度離散化してしまえば、これらの正方形上の値をどの順序で足し合わせてもよく、値が変わることを心配する必要はない。これは Fig. 25.5.3 に示されている。特に、

$$\sum_j \epsilon \left(\sum_i \epsilon f(\epsilon i, \epsilon j)\right). \tag{25.5.29}$$

と書ける。

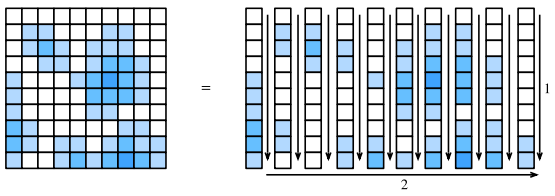


Fig. 25.5.3 多数の正方形にわたる和を、まず列ごとの和 (1) に分解し、その列和をさらに足し合わせる (2) ことを示す図。

内側の和は、まさに次の積分の離散化である。

$$G(\epsilon j) = \int_a^b f(x, \epsilon j)\,dx. \tag{25.5.30}$$

最後に、この 2 つの式を組み合わせると、

$$\sum_j \epsilon G(\epsilon j) \approx \int_c^d G(y)\,dy = \int_{[a,b]\times[c,d]} f(x,y)\,dx\,dy. \tag{25.5.31}$$

したがって、まとめると

$$\int_{[a,b]\times[c,d]} f(x,y)\,dx\,dy = \int_c^d \left(\int_a^b f(x,y)\,dx\right) dy. \tag{25.5.32}$$

一度離散化してしまえば、私たちがやったことは数の列を足す順序を入れ替えただけだということに注意する。すると、これは大したことではないように思えるかもしれない。しかし、この結果（フビニの定理と呼ばれる）は常に成り立つわけではない！機械学習で扱う種類の数学（連続関数）では問題ないが、成り立たない例を作ることも可能である（たとえば、長方形 $[0,2]\times[0,1]$ 上の関数 $f(x,y) = xy(x^2-y^2)/(x^2+y^2)^3$ など）。

x を先に積分し、その後 y を積分するという選択は任意であったことに注意する。同様に、先に y、次に x を積分して

$$\int_{[a,b]\times[c,d]} f(x,y)\; dx\; dy = \int_a^b \left(\int_c^d f(x,y)\; dy \right) dx. \tag{25.5.33}$$

と書くこともできる。

しばしば、ベクトル表記にまとめて、$U = [a,b]\times[c,d]$ に対して

$$\int_U f(\mathbf{x})\; d\mathbf{x}. \tag{25.5.34}$$

と書く。

25.5.6 多重積分における変数変換

(25.5.18) の 1 変数の場合と同様に、高次元の積分の中で変数を変えることは重要な道具である。ここでは導出は省略して、結果だけをまとめる。

積分領域を再パラメータ化する関数が必要である。これを $\phi : \mathbb{R}^n \to \mathbb{R}^n$ とする。つまり、n 個の実変数を受け取り、別の n 個を返す任意の関数である。式をすっきりさせるために、ϕ は 単射、すなわち自分自身に折り返すことがない（$\phi(\mathbf{x}) = \phi(\mathbf{y}) \implies \mathbf{x} = \mathbf{y}$）と仮定する。

このとき、

$$\int_{\phi(U)} f(\mathbf{x})\; d\mathbf{x} = \int_U f(\phi(\mathbf{x}))\, |\det(D\phi(\mathbf{x}))|\; d\mathbf{x}. \tag{25.5.35}$$

と書ける。

ここで $D\phi$ は ϕ の ヤコビアンであり、$\boldsymbol{\phi} = (\phi_1(x_1,\dots,x_n),\dots,\phi_n(x_1,\dots,x_n))$ の偏微分からなる行列である。

$$D\boldsymbol{\phi} = \begin{bmatrix} \frac{\partial \phi_1}{\partial x_1} & \cdots & \frac{\partial \phi_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \phi_n}{\partial x_1} & \cdots & \frac{\partial \phi_n}{\partial x_n} \end{bmatrix}. \tag{25.5.36}$$

よく見ると、これは 1 変数の連鎖律 (25.5.18) に似ているが、$\frac{du}{dx}(x)$ の代わりに $|\det(D\phi(\mathbf{x}))|$ が現れている点が異なる。この項をどう解釈すればよいか見てみよう。$\frac{du}{dx}(x)$ の項は、u を適用することで x 軸がどれだけ伸びるかを表していた。高次元で同じことを行うには、$\boldsymbol{\phi}$ を適用したときに、小さな正方形（あるいは小さな 超立方体）の面積（あるいは体積、あるいは超体積）がどれだけ伸びるか

を調べる。もし $\boldsymbol{\phi}$ が行列による乗算であれば、行列式がその答えを与えることはすでに知っている。

少し工夫すれば、ヤコビアンは、微分や勾配を用いて直線や平面で近似できるのと同じように、多変数関数 $\boldsymbol{\phi}$ をある点の近くで最もよく近似する行列であることが示せる。したがって、ヤコビアンの行列式は、1 次元で見たスケーリング因子をそのまま高次元に拡張したものになっている。

これを厳密に埋めるには少し手間がかかるので、今すぐ完全に明確でなくても心配はいりない。少なくとも、後で使う例を 1 つ見てみよう。次の積分を考える。

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-x^2-y^2}\,dx\,dy. \tag{25.5.37}$$

この積分を直接いじっても先には進めないが、変数変換を使えば大きく前進できる。$\boldsymbol{\phi}(r,\theta) = (r\cos(\theta), r\sin(\theta))$（つまり $x = r\cos(\theta)$、$y = r\sin(\theta)$）とおくと、変数変換公式により、これは次と同じである。

$$\int_{0}^{\infty}\int_{0}^{2\pi} e^{-r^2}\,|\det(D\boldsymbol{\phi}(\mathbf{x}))|\,d\theta\,dr, \tag{25.5.38}$$

ここで

$$|\det(D\boldsymbol{\phi}(\mathbf{x}))| = \left|\det\begin{bmatrix}\cos(\theta) & -r\sin(\theta)\\ \sin(\theta) & r\cos(\theta)\end{bmatrix}\right| = r(\cos^2(\theta)+\sin^2(\theta)) = r. \tag{25.5.39}$$

したがって、積分は

$$\int_{0}^{\infty}\int_{0}^{2\pi} re^{-r^2}\,d\theta\,dr = 2\pi\int_{0}^{\infty} re^{-r^2}\,dr = \pi, \tag{25.5.40}$$

となる。最後の等号は、Section 25.5.3 節で使ったのと同じ計算に従う。

この積分は、Section 25.6 で連続確率変数を学ぶときに再び登場する。

25.5.7 まとめ

- 積分の理論により、面積や体積に関する問いに答えられる。
- 微積分の基本定理により、ある点までの面積の導関数が積分される関数の値で与えられるという観察を通じて、導関数に関する知識を使って面積を計算できる。
- 高次元の積分は、1 変数積分を繰り返すことで計算できる。

25.5.8 演習

1. $\int_1^2 \frac{1}{x}\,dx$ はいくらか？
2. 変数変換公式を用いて、$\int_0^{\sqrt{\pi}} x\sin(x^2)\,dx$ を積分せよ。
3. $\int_{[0,1]^2} xy\,dx\,dy$ はいくらか？

4. 変数変換公式を用いて $\int_0^2 \int_0^1 xy(x^2 - y^2)/(x^2 + y^2)^3 \; dy \; dx$ と $\int_0^1 \int_0^2 f(x, y) = xy(x^2 - y^2)/(x^2 + y^2)^3 \; dx \; dy$ を計算し、それらが異なることを確かめよ。

25.6 確率変数

Section 5.6 では、離散確率変数の扱い方の基礎を見た。ここでいう離散確率変数とは、有限個の取りうる値、または整数値をとる確率変数のことでした。この節では、任意の実数値をとりうる 連続確率変数の理論を展開する。

25.6.1 連続確率変数

連続確率変数は、離散確率変数よりもかなり繊細な話題である。技術的な飛躍としては、数のリストを足し合わせることから関数を積分することへの飛躍に匹敵すると考えるとよいだろう。そのため、理論を構築するのに少し時間をかける必要がある。

離散から連続へ

連続確率変数を扱う際に生じる追加の技術的課題を理解するために、思考実験をしてみよう。ダーツをダーツボードに投げて、ボードの中心からちょうど 2cm の位置に当たる確率を知りたいとする。

まず、1 桁の精度で測ることを考える。つまり、0cm、1cm、2cm などのビンに分けるのである。たとえばダーツを 100 本投げて、そのうち 20 本が 2cm のビンに入ったなら、投げたダーツの 20% が中心から 2cm の位置に当たったと結論づける。

しかし、よく見ると、これは私たちの問いと一致していない。私たちが知りたかったのは「ちょうど」当たる確率であって、これらのビンはたとえば 1.5cm から 2.5cm の間に入ったものをすべて含んでいるからである。

それでもあきらめず、さらに進む。今度は 1.9cm、2.0cm、2.1cm のように、もっと精密に測りる。すると、100 本のうち 3 本が 2.0cm のビンに入ったとしよう。そこで確率は 3% だと結論づける。

しかし、これでも何も解決していない。問題を 1 桁先送りしただけである。少し抽象化してみよう。最初の k 桁が $2.00000\ldots$ と一致する確率がわかっていて、最初の $k+1$ 桁まで一致する確率を知りたいとする。$k+1^{\text{th}}$ 桁は、集合 $\{0, 1, 2, \ldots, 9\}$ からほぼランダムに選ばれると考えるのはかなり自然である。少なくとも、中心から何マイクロメートル離れているかの末尾が 7 になるか 3 になるかを強制するような、物理的に意味のある過程は思いつかない。

つまり、本質的には、精度を 1 桁上げるごとに一致する確率は 10 分の 1 に減る

はずである。言い換えると、次のように期待できる。

$$P(\text{distance is } 2.00\ldots, \text{ to } k \text{ digits}) \approx p \cdot 10^{-k}. \tag{25.6.1}$$

ここでの値 p は、最初の数桁で何が起こるかを実質的に表し、10^{-k} は残りを扱う。

小数点以下 $k = 4$ 桁まで正確に位置がわかるということは、値がたとえば $[1.99995, 2.00005]$ のような区間に入ることを意味する。この区間の長さは $2.00005 - 1.99995 = 10^{-4}$ である。したがって、この区間の長さを ϵ と呼ぶと、

$$P(\text{distance is in an } \epsilon\text{-sized interval around } 2) \approx \epsilon \cdot p. \tag{25.6.2}$$

これをさらに一歩進めよう。ここまでずっと点 2 について考えてきたが、他の点については考えていなかった。そこでも本質的には何も違いませんが、値 p はおそらく異なる。少なくとも、ダーツ投げでは 20cm より 2cm のように中心に近い点に当たりやすいと期待したいところである。したがって、値 p は固定ではなく、点 x に依存すべきである。すると、次を期待すべきだとわかる。

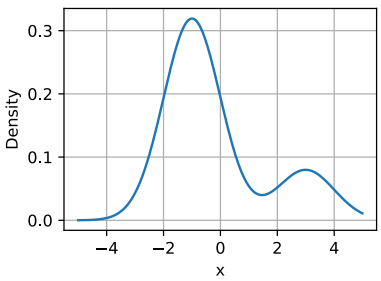
$$P(\text{distance is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(x). \tag{25.6.3}$$

実際、(25.6.3) は 確率密度関数を正確に定義している。これは、ある点の近くに当たる相対的な確率を表す関数 $p(x)$ である。このような関数がどのような形をしているかを可視化してみよう。

```
%matplotlib inline
from d2l import torch as d2l
from IPython import display
import torch
torch.pi = torch.acos(torch.zeros(1)).item() * 2  # Define pi in torch

# Plot the probability density function for some random variable
x = torch.arange(-5, 5, 0.01)
p = 0.2*torch.exp(-(x - 3)**2 / 2)/torch.sqrt(2 * torch.tensor(torch.pi)) + \
    0.8*torch.exp(-(x + 1)**2 / 2)/torch.sqrt(2 * torch.tensor(torch.pi))

d2l.plot(x, p, 'x', 'Density')
```

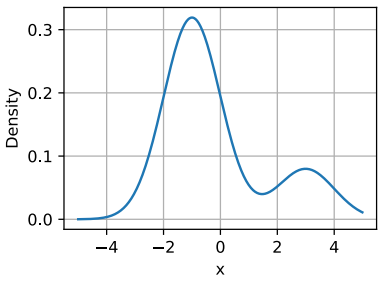


```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from mxnet import np, npx
npx.set_np()

# Plot the probability density function for some random variable
x = np.arange(-5, 5, 0.01)
p = 0.2*np.exp(-(x - 3)**2 / 2)/np.sqrt(2 * np.pi) + \
    0.8*np.exp(-(x + 1)**2 / 2)/np.sqrt(2 * np.pi)

d2l.plot(x, p, 'x', 'Density')
```

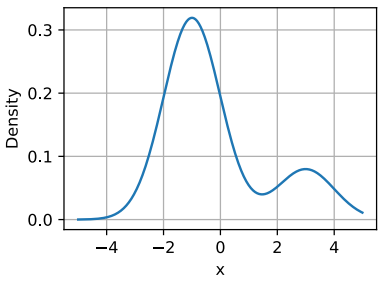
```
[07:03:01] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```




```
%matplotlib inline
from d2l import tensorflow as d2l
from IPython import display
import tensorflow as tf
tf.pi = tf.acos(tf.zeros(1)).numpy() * 2  # Define pi in TensorFlow

# Plot the probability density function for some random variable
x = tf.range(-5, 5, 0.01)
p = 0.2*tf.exp(-(x - 3)**2 / 2)/tf.sqrt(2 * tf.constant(tf.pi)) + \
    0.8*tf.exp(-(x + 1)**2 / 2)/tf.sqrt(2 * tf.constant(tf.pi))

d2l.plot(x, p, 'x', 'Density')
```



関数値が大きい場所は、確率変数が見つかりやすい領域を示している。値が低い部分は、確率変数が見つかりにくい領域である。

確率密度関数

では、これをさらに詳しく調べよう。確率変数 X に対する確率密度関数とは何かをすでに直感的に見た。すなわち、密度関数とは

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(x). \tag{25.6.4}$$

を満たす関数 $p(x)$ のことである。

では、これは $p(x)$ の性質について何を意味するのだろうか。

まず、確率は負にならないので、$p(x) \geq 0$ であると期待される。

次に、$\mathbb{R}$ を幅 ϵ の無限個の区間、たとえば $(\epsilon \cdot i, \epsilon \cdot (i+1)]$ に分割することを考える。それぞれについて、(25.6.4) より確率はおおよそ

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(\epsilon \cdot i), \tag{25.6.5}$$

だろうから、すべてを足し合わせると

$$P(X \in \mathbb{R}) \approx \sum_i \epsilon \cdot p(\epsilon \cdot i). \tag{25.6.6}$$

これは Section 25.5 で議論した積分の近似にほかならない。したがって、

$$P(X \in \mathbb{R}) = \int_{-\infty}^{\infty} p(x)\, dx. \tag{25.6.7}$$

確率変数は 何らかの値をとらなければならないので、$P(X \in \mathbb{R}) = 1$ である。したがって、任意の密度について

$$\int_{-\infty}^{\infty} p(x)\, dx = 1. \tag{25.6.8}$$

実際、さらに掘り下げると、任意の a と b について

$$P(X \in (a, b]) = \int_a^b p(x)\, dx. \tag{25.6.9}$$

が成り立ちる。

これをコードで近似するには、先ほどと同じ離散近似法を使える。この場合、青い領域に入る確率を近似できる。

```
# Approximate probability using numerical integration
epsilon = 0.01
x = torch.arange(-5, 5, 0.01)
p = 0.2*torch.exp(-(x - 3)**2 / 2) / torch.sqrt(2 * torch.tensor(torch.pi)) +\
    0.8*torch.exp(-(x + 1)**2 / 2) / torch.sqrt(2 * torch.tensor(torch.pi))

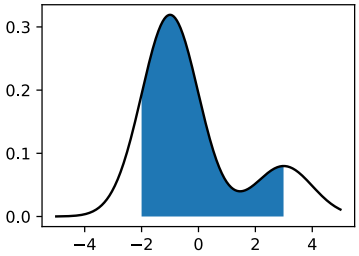
d2l.set_figsize()
d2l.plt.plot(x, p, color='black')
```

(continues on next page)

(continued from previous page)

```
d2l.plt.fill_between(x.tolist()[300:800], p.tolist()[300:800])
d2l.plt.show()

f'approximate Probability: {torch.sum(epsilon*p[300:800])}'
```

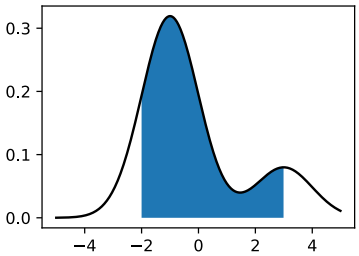



```
'approximate Probability: 0.7736171483993353'
```

```
# Approximate probability using numerical integration
epsilon = 0.01
x = np.arange(-5, 5, 0.01)
p = 0.2*np.exp(-(x - 3)**2 / 2) / np.sqrt(2 * np.pi) + \
    0.8*np.exp(-(x + 1)**2 / 2) / np.sqrt(2 * np.pi)

d2l.set_figsize()
d2l.plt.plot(x, p, color='black')
d2l.plt.fill_between(x.tolist()[300:800], p.tolist()[300:800])
d2l.plt.show()

f'approximate Probability: {np.sum(epsilon*p[300:800])}'
```




```
'approximate Probability: 0.7736172080039978'
```

```
# Approximate probability using numerical integration
epsilon = 0.01
x = tf.range(-5, 5, 0.01)
p = 0.2*tf.exp(-(x - 3)**2 / 2) / tf.sqrt(2 * tf.constant(tf.pi)) +\
```

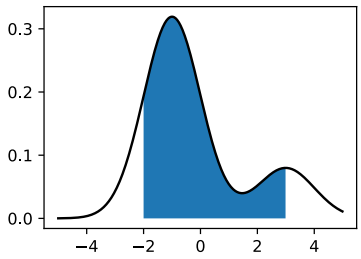
(continues on next page)

(continued from previous page)

```
    0.8*tf.exp(-(x + 1)**2 / 2) / tf.sqrt(2 * tf.constant(tf.pi))

d2l.set_figsize()
d2l.plt.plot(x, p, color='black')
d2l.plt.fill_between(x.numpy().tolist()[300:800], p.numpy().tolist()[300:800])
d2l.plt.show()

f'approximate Probability: {tf.reduce_sum(epsilon*p[300:800])}'
```




```
'approximate Probability: 0.7736172080039978'
```

これら 2 つの性質は、可能な確率密度関数の空間をちょうど記述している（よく使われる略称として *p.d.f.* とも呼ばれます）。すなわち、非負関数 $p(x) \geq 0$ であって

$$\int_{-\infty}^{\infty} p(x)\ dx = 1. \tag{25.6.10}$$

を満たすものである。

この関数は、確率変数が特定の区間に入る確率を積分によって求めることで解釈する。

$$P(X \in (a, b]) = \int_{a}^{b} p(x)\ dx. \tag{25.6.11}$$

Section 25.8 では、いくつかの代表的な分布を見ますが、ここでは抽象的な議論を続けましょう。

累積分布関数

前節では p.d.f. の概念を見た。実際には、これは連続確率変数を扱う際によく使われる方法であるが、1 つ大きな落とし穴がある。それは、p.d.f. の値そのものは確率ではなく、確率を得るために積分しなければならない関数だということである。密度が 10 より大きくても、それが長さ 1/10 を超える区間にわたって大きくなければ問題ない。これは直感に反することがあるため、人々はしばしば累積分布関数、すなわち c.d.f. でも考える。こちらは 確率である。

特に、(25.6.11) を用いて、密度 $p(x)$ をもつ確率変数 X の c.d.f. を

$$F(x) = \int_{-\infty}^{x} p(x)\,dx = P(X \le x). \tag{25.6.12}$$

と定義する。

いくつかの性質を見てみよう。

- $x \to -\infty$ のとき $F(x) \to 0$。
- $x \to \infty$ のとき $F(x) \to 1$。
- $F(x)$ は単調非減少である（$y > x \implies F(y) \ge F(x)$）。
- X が連続確率変数なら、$F(x)$ は連続である（飛びがない）。

4 つ目について、X が離散的、たとえば 0 と 1 をそれぞれ確率 1/2 でとる場合には、これは成り立たないことに注意する。その場合、

$$F(x) = \begin{cases} 0 & x < 0, \\ \frac{1}{2} & x < 1, \\ 1 & x \ge 1. \end{cases} \tag{25.6.13}$$

この例から、c.d.f. を使う利点の 1 つがわかる。連続確率変数と離散確率変数を同じ枠組みで扱えること、さらには両者の混合（コインを投げて、表ならサイコロの目を返し、裏ならダーツボードの中心からの距離を返す）も扱えることである。

平均

確率変数 X を扱っているとしよう。分布そのものは解釈が難しいことがある。確率変数の振る舞いを簡潔に要約できると便利なことがよくある。確率変数の振る舞いを捉えるのに役立つ数を 要約統計量と呼ぶ。最もよく出てくるものは平均、分散、標準偏差である。

平均は確率変数の平均値を表する。離散確率変数 X が値 x_i を確率 p_i でとるなら、平均は重み付き平均で与えられる。すなわち、値にその値をとる確率を掛けて足し合わせる。

$$\mu_X = E[X] = \sum_i x_i p_i. \tag{25.6.14}$$

平均の解釈は、注意は必要であるが、本質的には確率変数がどのあたりに位置しがちかを示すものだと考えればよいだろう。

この節を通して扱う最小限の例として、X を、確率 p で $a-2$、確率 p で $a+2$、確率 $1-2p$ で a をとる確率変数とする。(25.6.14) を使うと、a と p のどのような選び方に対しても平均は

$$\mu_X = E[X] = \sum_i x_i p_i = (a-2)p + a(1-2p) + (a+2)p = a. \tag{25.6.15}$$

となる。したがって、平均は a である。これは、確率変数を中心化した位置が a であるという直感と一致している。

役に立つので、いくつかの性質をまとめておきよう。

- 任意の確率変数 X と数 a, b に対して、$\mu_{aX+b} = a\mu_X + b$ である。
- 2 つの確率変数 X と Y に対して、$\mu_{X+Y} = \mu_X + \mu_Y$ である。

平均は確率変数の平均的な振る舞いを理解するのに有用であるが、平均だけでは十分な直感的理解には至りない。1 回の販売あたり \$10 ± \$1 の利益を得るのと、\$10 ± \$15 の利益を得るのとでは、平均値が同じでもまったく違いる。後者のほうが変動がはるかに大きく、したがってリスクもはるかに大きいのである。したがって、確率変数の振る舞いを理解するには、少なくともう 1 つ、確率変数がどれだけ広く変動するかを測る尺度が必要である。

分散

そこで、確率変数の 分散を考える。これは、確率変数が平均からどれだけずれるかを定量的に測るものである。式 $X - \mu_X$ を考えよう。これは確率変数の平均からの偏差である。この値は正にも負にもなりうるので、偏差の大きさを測るために正にする工夫が必要である。

試してみる自然な方法は $|X - \mu_X|$ を見ることである。実際、これは 平均絶対偏差と呼ばれる有用な量につながりるが、数学や統計の他分野とのつながりのため、しばしば別の方法が使われる。

具体的には、$(X - \mu_X)^2$ を見る。この量の典型的な大きさを平均で見れば、分散に到達する。

$$\sigma_X^2 = \mathrm{Var}(X) = E\left[(X - \mu_X)^2\right] = E[X^2] - \mu_X^2. \tag{25.6.16}$$

(25.6.16) の最後の等式は、中央の定義を展開し、期待値の性質を適用することで得られる。

例として、X が確率 p で $a-2$、確率 p で $a+2$、確率 $1-2p$ で a をとる確率変数を考えよう。この場合 $\mu_X = a$ なので、計算すべきなのは $E\left[X^2\right]$ だけである。これはすぐに求められる。

$$E\left[X^2\right] = (a-2)^2 p + a^2(1-2p) + (a+2)^2 p = a^2 + 8p. \tag{25.6.17}$$

したがって、(25.6.16) より分散は

$$\sigma_X^2 = \mathrm{Var}(X) = E[X^2] - \mu_X^2 = a^2 + 8p - a^2 = 8p. \tag{25.6.18}$$

この結果もまた納得できる。p の最大値は $1/2$ であり、これは $a-2$ か $a+2$ をコイン投げで選ぶ場合に対応する。このとき分散が 4 になるのは、$a-2$ と $a+2$ のどちらも平均から 2 だけ離れており、$2^2 = 4$ だからである。逆に、$p = 0$ ならこの確率変数は常に 0 をとるので、分散はまったくない。

分散の性質をいくつか挙げておきる。

- 任意の確率変数 X に対して、$\mathrm{Var}(X) \geq 0$ であり、$\mathrm{Var}(X) = 0$ となるのは X が定数であるとき、かつそのときに限る。
- 任意の確率変数 X と数 a, b に対して、$\mathrm{Var}(aX + b) = a^2\mathrm{Var}(X)$ である。
- 2 つの 独立な確率変数 X と Y に対して、$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$ である。

これらの値を解釈する際には、少し引っかかる点がある。特に、単位を追跡しながらこの計算がどうなるかを考えてみよう。ウェブページ上で製品に付けられた星評価を扱っているとする。このとき a、$a-2$、$a+2$ はすべて星の単位で測られる。同様に、平均 μ_X も（重み付き平均なので）星の単位である。しかし分散に進むと、すぐに問題にぶつかりる。$(X - \mu_X)^2$ を見る必要があり、これは 二乗された星の単位になるからである。つまり、分散そのものは元の測定値と比較しにくいのである。解釈可能にするには、元の単位に戻る必要がある。

標準偏差

この要約統計量は、平方根を取ることで分散から常に導ける。そこで 標準偏差を

$$\sigma_X = \sqrt{\mathrm{Var}(X)}. \tag{25.6.19}$$

と定義する。

この例では、標準偏差は $\sigma_X = 2\sqrt{2p}$ になる。レビューの例で単位が星なら、σ_X も再び星の単位である。

分散について述べた性質は、標準偏差について次のように言い換えられる。

- 任意の確率変数 X に対して、$\sigma_X \geq 0$。
- 任意の確率変数 X と数 a, b に対して、$\sigma_{aX+b} = |a|\sigma_X$。
- 2 つの 独立な確率変数 X と Y に対して、$\sigma_{X+Y} = \sqrt{\sigma_X^2 + \sigma_Y^2}$。

ここで、「標準偏差は元の確率変数と同じ単位を持つのなら、その確率変数について何か描けるものを表しているのだろうか？」と問うのは自然である。答えは大いにイエスである。平均が確率変数の典型的な位置を示したのと同様に、標準偏差はその確率変数の典型的な変動範囲を与える。これはチェビシェフの不等式として知られるものによって厳密にできる。

$$P\left(X \notin [\mu_X - \alpha\sigma_X, \mu_X + \alpha\sigma_X]\right) \leq \frac{1}{\alpha^2}. \tag{25.6.20}$$

言葉で言えば、$\alpha = 10$ の場合、どんな確率変数でも標本の 99% は平均から 10 標準偏差以内に入る。これにより、標準的な要約統計量の解釈がすぐに得られる。

この主張がかなり繊細であることを見るために、先ほどの例、すなわち X が確率 p で $a-2$、確率 p で $a+2$、確率 $1-2p$ で a をとる確率変数をもう一度見てみ

よう。平均は a、標準偏差は $2\sqrt{2p}$ でした。したがって、チェビシェフの不等式 (25.6.20) を $\alpha = 2$ で使うと、

$$P\left(X \notin [a - 4\sqrt{2p}, a + 4\sqrt{2p}]\right) \leq \frac{1}{4}. \tag{25.6.21}$$

となる。つまり、どんな p に対しても、この確率変数は 75% の確率でこの区間に入る。ここで、$p \to 0$ とすると、この区間は単一点 a に収束する。しかし、確率変数は $a-2, a, a+2$ の値しかとらないので、最終的には $a-2$ と $a+2$ が区間の外に出ることは確実である。問題は、それがどの p で起こるかである。そこで、$a + 4\sqrt{2p} = a + 2$ となる p を解くと、$p = 1/8$ であり、これは分布からの標本の 1/4 以下しか区間の外に出ないという主張に反しないまま、それが起こりうる最初の p です（左に 1/8、右に 1/8）。

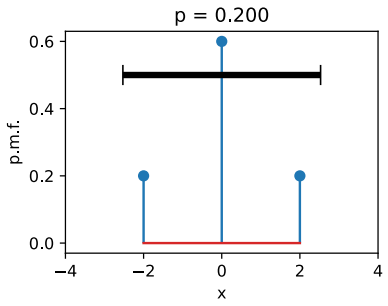
これを可視化してみよう。3 つの値が出る確率を、それぞれ確率に比例した高さの 3 本の縦棒で示する。区間は中央に水平線として描く。最初の図は、$p > 1/8$ のとき、区間がすべての点を安全に含む様子を示する。

```
# Define a helper to plot these figures
def plot_chebyshev(a, p):
    d2l.set_figsize()
    d2l.plt.stem([a-2, a, a+2], [p, 1-2*p, p], use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')

    d2l.plt.hlines(0.5, a - 4 * torch.sqrt(2 * p),
                   a + 4 * torch.sqrt(2 * p), 'black', lw=4)
    d2l.plt.vlines(a - 4 * torch.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.vlines(a + 4 * torch.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.title(f'p = {p:.3f}')

    d2l.plt.show()

# Plot interval when p > 1/8
plot_chebyshev(0.0, torch.tensor(0.2))
```




```
# Define a helper to plot these figures
def plot_chebyshev(a, p):
```

(continues on next page)

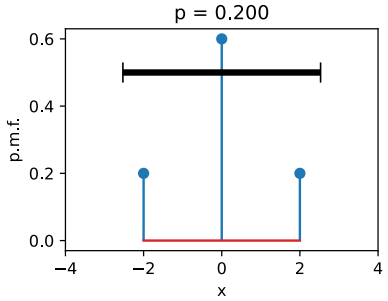
(continued from previous page)

```
    d2l.set_figsize()
    d2l.plt.stem([a-2, a, a+2], [p, 1-2*p, p], use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')

    d2l.plt.hlines(0.5, a - 4 * np.sqrt(2 * p),
                   a + 4 * np.sqrt(2 * p), 'black', lw=4)
    d2l.plt.vlines(a - 4 * np.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.vlines(a + 4 * np.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.title(f'p = {p:.3f}')

    d2l.plt.show()

# Plot interval when p > 1/8
plot_chebyshev(0.0, 0.2)
```



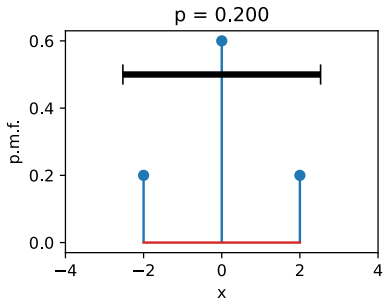

```
# Define a helper to plot these figures
def plot_chebyshev(a, p):
    d2l.set_figsize()
    d2l.plt.stem([a-2, a, a+2], [p, 1-2*p, p], use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')

    d2l.plt.hlines(0.5, a - 4 * tf.sqrt(2 * p),
                   a + 4 * tf.sqrt(2 * p), 'black', lw=4)
    d2l.plt.vlines(a - 4 * tf.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.vlines(a + 4 * tf.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.title(f'p = {p:.3f}')

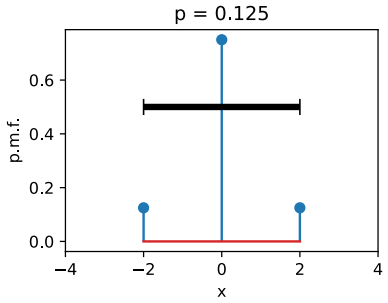
    d2l.plt.show()

# Plot interval when p > 1/8
plot_chebyshev(0.0, tf.constant(0.2))
```

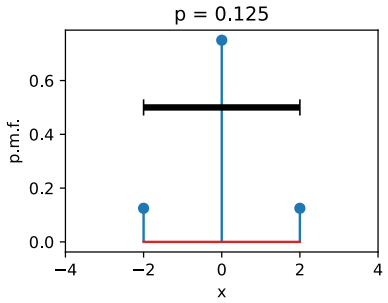
2 つ目は、$p = 1/8$ のとき、区間がちょうど 2 点に接することを示する。これは、不等式を真に保ちながらこれ以上小さい区間を取れないので、不等式が 鋭いことを示している。




```
# Plot interval when p = 1/8
plot_chebyshev(0.0, torch.tensor(0.125))
```

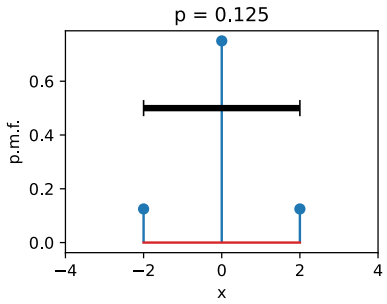



```
# Plot interval when p = 1/8
plot_chebyshev(0.0, 0.125)
```



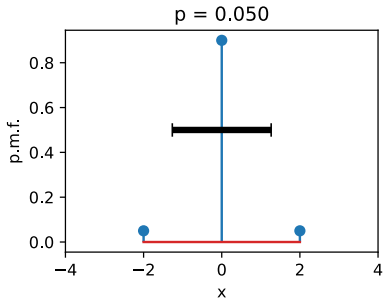

```
# Plot interval when p = 1/8
plot_chebyshev(0.0, tf.constant(0.125))
```

3 つ目は、$p < 1/8$ のとき、区間が中心だけを含むことを示する。これは不等式を破るものではない。なぜなら、区間の外に出る確率が $1/4$ を超えないことだ

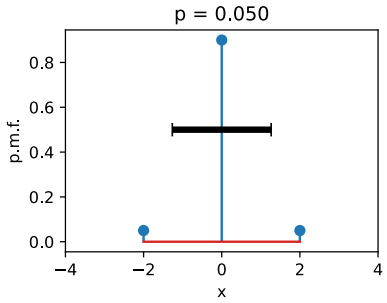


けを保証すればよいからである。したがって、$p < 1/8$ になれば、$a-2$ と $a+2$ の 2 点は捨ててよいのである。

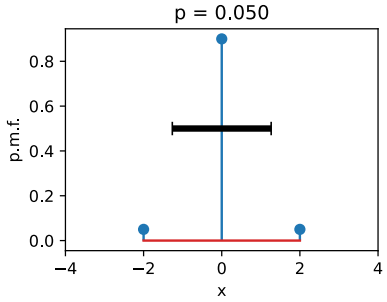
```
# Plot interval when p < 1/8
plot_chebyshev(0.0, torch.tensor(0.05))
```




```
# Plot interval when p < 1/8
plot_chebyshev(0.0, 0.05)
```




```
# Plot interval when p < 1/8
plot_chebyshev(0.0, tf.constant(0.05))
```



連続体における平均と分散

ここまでは離散確率変数について述べてきたが、連続確率変数の場合も同様である。これを直感的に理解するために、実数直線を長さ ϵ の区間 $(\epsilon i, \epsilon(i+1)]$ に分割することを考えよう。そうすると、連続確率変数は離散化され、(25.6.14) を使って

$$\begin{aligned} \mu_X &\approx \sum_i (\epsilon i) P(X \in (\epsilon i, \epsilon(i+1)]) \\ &\approx \sum_i (\epsilon i) p_X(\epsilon i) \epsilon, \end{aligned} \tag{25.6.22}$$

と書ける。ここで p_X は X の密度である。これは $xp_X(x)$ の積分の近似なので、

$$\mu_X = \int_{-\infty}^{\infty} x p_X(x) \, dx. \tag{25.6.23}$$

と結論できる。

同様に、(25.6.16) を使うと分散は

$$\sigma_X^2 = E[X^2] - \mu_X^2 = \int_{-\infty}^{\infty} x^2 p_X(x) \, dx - \left(\int_{-\infty}^{\infty} x p_X(x) \, dx \right)^2. \tag{25.6.24}$$

と書ける。

平均、分散、標準偏差について上で述べたことは、この場合にもすべて当てはまりる。たとえば、密度

$$p(x) = \begin{cases} 1 & x \in [0, 1], \\ 0 & \text{otherwise.} \end{cases} \tag{25.6.25}$$

をもつ確率変数を考えると、

$$\mu_X = \int_{-\infty}^{\infty} x p(x) \, dx = \int_0^1 x \, dx = \frac{1}{2}. \tag{25.6.26}$$

また

$$\sigma_X^2 = \int_{-\infty}^{\infty} x^2 p(x) \, dx - \left(\frac{1}{2} \right)^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}. \tag{25.6.27}$$

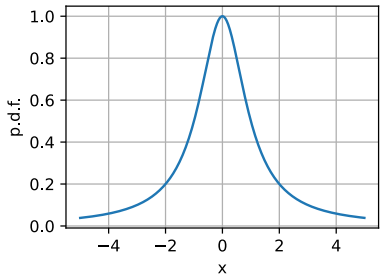
警告として、もう 1 つの例、コーシー分布を見てみよう。これは p.d.f. が

$$p(x) = \frac{1}{1+x^2}. \tag{25.6.28}$$

で与えられる分布である。

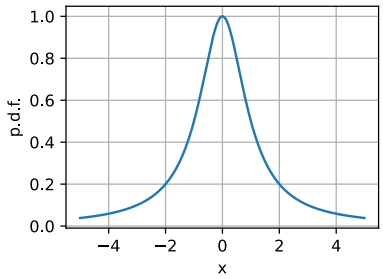
```
# Plot the Cauchy distribution p.d.f.
x = torch.arange(-5, 5, 0.01)
p = 1 / (1 + x**2)

d2l.plot(x, p, 'x', 'p.d.f.')
```




```
# Plot the Cauchy distribution p.d.f.
x = np.arange(-5, 5, 0.01)
p = 1 / (1 + x**2)

d2l.plot(x, p, 'x', 'p.d.f.')
```

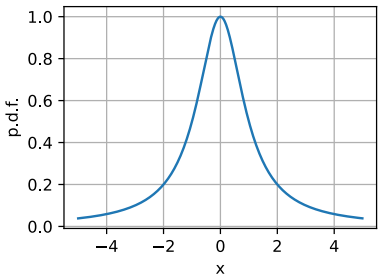



```
# Plot the Cauchy distribution p.d.f.
x = tf.range(-5, 5, 0.01)
p = 1 / (1 + x**2)

d2l.plot(x, p, 'x', 'p.d.f.')
```

この関数は無害に見え、実際、積分表を参照すると下の面積が 1 であることがわかるので、連続確率変数を定義している。

何がまずいのかを見るために、これの分散を計算してみよう。これは (25.6.16)



を使って

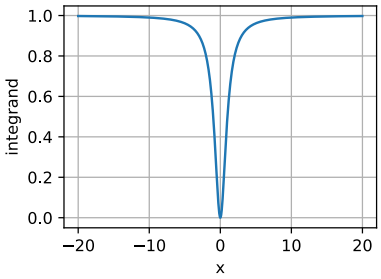
$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2}\,dx \tag{25.6.29}$$

を計算することに相当する。

中の関数は次のようになる。

```
# Plot the integrand needed to compute the variance
x = torch.arange(-20, 20, 0.01)
p = x**2 / (1 + x**2)

d2l.plot(x, p, 'x', 'integrand')
```



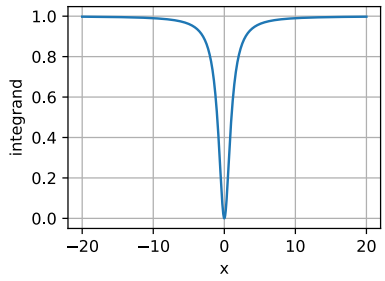

```
# Plot the integrand needed to compute the variance
x = np.arange(-20, 20, 0.01)
p = x**2 / (1 + x**2)

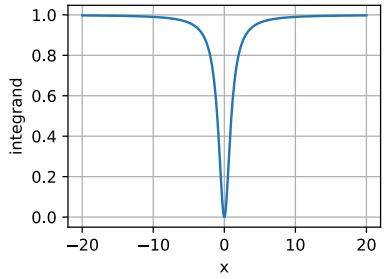
d2l.plot(x, p, 'x', 'integrand')
```

```
# Plot the integrand needed to compute the variance
x = tf.range(-20, 20, 0.01)
p = x**2 / (1 + x**2)

d2l.plot(x, p, 'x', 'integrand')
```

この関数は、ほぼ定数 1 であり、0 付近に小さなくぼみがあるだけなので、明ら





かに下の面積は無限大である。実際、

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2}\, dx = \infty. \tag{25.6.30}$$

と示せる。つまり、有限で定義された分散を持たない。

しかし、さらに深く見ると、もっと気がかりな結果が出る。(25.6.14) を使って平均を計算してみよう。変数変換の公式を使うと、

$$\mu_X = \int_{-\infty}^{\infty} \frac{x}{1+x^2}\, dx = \frac{1}{2}\int_{1}^{\infty} \frac{1}{u}\, du. \tag{25.6.31}$$

となる。右辺の積分は対数の定義なので、これは本質的には $\log(\infty) = \infty$ であり、平均値も定義できない。

機械学習の研究者は、通常はこうした問題を扱わなくて済むようにモデルを設計し、ほとんどの場合、平均と分散がきちんと定義された確率変数を扱う。しかし、時折、*heavy tails*（大きな値をとる確率が十分大きく、平均や分散のような量が定義できなくなる確率変数）をもつ確率変数が物理系のモデリングに役立つことがある。そのようなものが存在することは知っておく価値がある。

同時密度関数

ここまでの議論は、1 つの実数値確率変数を扱う場合を仮定していた。しかし、強く相関した 2 つ以上の確率変数を扱う場合はどうだろうか。この状況は機械学習では普通である。たとえば、画像の (i, j) 座標にある画素の赤成分を表す $R_{i,j}$ や、時刻 t における株価を表す確率変数 P_t を考えてみよ。近くの画素は似

た色を持ち、近い時刻の価格は似た値になりがちである。これらを別々の確率変数として扱い、その上で成功するモデルを作れるとは期待できない（Section 25.9 では、そのような仮定のために性能が劣るモデルを見ることになる）。相関した連続確率変数を扱うための数学的言語を整備する必要がある。

幸い、Section 25.5 の多重積分を使えば、そのような言語を構築できる。簡単のため、相関しうる 2 つの確率変数 X, Y があるとしよう。このとき、1 変数の場合と同様に、次の問いを考えられる。

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x \text{ and } Y \text{ is in an } \epsilon\text{-sized interval around } y). \tag{25.6.32}$$

1 変数の場合と同じ推論により、これはおおよそ

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x \text{ and } Y \text{ is in an } \epsilon\text{-sized interval around } y) \approx \epsilon^2 p(x, y), \tag{25.6.33}$$

となるはずである。ここで $p(x, y)$ はある関数である。これは X と Y の同時密度と呼ばれる。1 変数の場合と同様に、次の性質が成り立ちる。

- $p(x, y) \geq 0$;
- $\int_{\mathbb{R}^2} p(x, y)\, dx\, dy = 1$;
- $P((X, Y) \in \mathcal{D}) = \int_{\mathcal{D}} p(x, y)\, dx\, dy$。

このようにして、複数の、しかも相関しうる確率変数を扱える。2 つより多くの確率変数を扱いたい場合は、$p(\mathbf{x}) = p(x_1, \ldots, x_n)$ と考えることで、多変量密度を任意の次元に拡張できる。非負であり、全積分が 1 であるという性質は同様に成り立ちる。

周辺分布

複数の変数を扱うとき、関係を無視して「この 1 つの変数はどのように分布しているのか」と知りたいことがよくある。このような分布を 周辺分布と呼ぶ。

具体的に、同時密度が $p_{X,Y}(x, y)$ で与えられる 2 つの確率変数 X, Y があるとしよう。ここでは、密度がどの確率変数に対するものかを示すために添字を使う。周辺分布を求めるとは、この関数から $p_X(x)$ を求めることである。

多くのことと同様に、何が成り立つべきかを考えるには直感的な図に戻るのが最善である。密度とは

$$P(X \in [x, x + \epsilon]) \approx \epsilon \cdot p_X(x). \tag{25.6.34}$$

を満たす関数 p_X だったことを思い出する。

ここには Y の記述がないが、与えられているのが $p_{X,Y}$ だけなら、何らかの形で Y を含める必要がある。まず、これは次と同じだとわかる。

$$P(X \in [x, x + \epsilon], \text{ and } Y \in \mathbb{R}) \approx \epsilon \cdot p_X(x). \tag{25.6.35}$$

この場合については密度から直接はわからないので、y 方向にも小区間に分割する必要がある。すると、次のように書ける。

$$\begin{aligned}\epsilon \cdot p_X(x) &\approx \sum_i P(X \in [x, x+\epsilon], \text{and } Y \in [\epsilon \cdot i, \epsilon \cdot (i+1)]) \\ &\approx \sum_i \epsilon^2 p_{X,Y}(x, \epsilon \cdot i).\end{aligned} \tag{25.6.36}$$

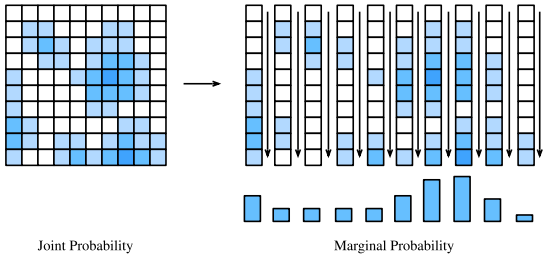


Fig. 25.6.1 確率の配列の列に沿って和をとることで、x 軸に対応する確率変数だけの周辺分布を得ることができる。

これは、Fig. 25.6.1 に示すように、密度の値を一列に並んだ正方形に沿って足し合わせることを意味する。実際、両辺から ϵ の因子を 1 つ打ち消し、右辺の和が y に関する積分であることを認識すると、

$$\begin{aligned}p_X(x) &\approx \sum_i \epsilon p_{X,Y}(x, \epsilon \cdot i) \\ &\approx \int_{-\infty}^{\infty} p_{X,Y}(x, y)\, dy.\end{aligned} \tag{25.6.37}$$

と結論できる。

したがって、

$$p_X(x) = \int_{-\infty}^{\infty} p_{X,Y}(x, y)\, dy. \tag{25.6.38}$$

となる。つまり、周辺分布を得るには、関心のない変数について積分すればよいのである。この過程は、しばしば不要な変数を 積分消去する、あるいは 周辺化すると呼ばれる。

共分散

複数の確率変数を扱うとき、知っておくと役立つもう 1 つの要約統計量が共分散である。これは、2 つの確率変数がどの程度一緒に変動するかを測りる。

2 つの確率変数 X と Y があるとする。まず、それらが離散的で、値 (x_i, y_j) を確率 p_{ij} でとるとしよう。この場合、共分散は

$$\sigma_{XY} = \mathrm{Cov}(X, Y) = \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y) p_{ij}. = E[XY] - E[X]E[Y]. \tag{25.6.39}$$

と定義される。

直感的に考えるために、次の確率変数の組を見てみよう。X は値 1 と 3 をとり、Y は値 -1 と 3 をとるとする。次の確率を持つとしよう。

$$\begin{aligned} P(X = 1 \text{ and } Y = -1) &= \frac{p}{2}, \\ P(X = 1 \text{ and } Y = 3) &= \frac{1-p}{2}, \\ P(X = 3 \text{ and } Y = -1) &= \frac{1-p}{2}, \\ P(X = 3 \text{ and } Y = 3) &= \frac{p}{2}, \end{aligned} \tag{25.6.40}$$

ここで p は $[0, 1]$ の範囲で選べるパラメータである。$p = 1$ なら、両者は常に同時に最小値または最大値をとり、$p = 0$ なら、同時に反転した値をとることが保証される（片方が大きいとき他方は小さく、その逆も同様です）。$p = 1/2$ なら 4 つの可能性はすべて同確率で、互いに関係がないはずである。共分散を計算してみよう。まず、$\mu_X = 2$、$\mu_Y = 1$ なので、(25.6.39) を使って

$$\begin{aligned} \mathrm{Cov}(X, Y) &= \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y) p_{ij} \\ &= (1-2)(-1-1)\frac{p}{2} + (1-2)(3-1)\frac{1-p}{2} + (3-2)(-1-1)\frac{1-p}{2} + (3-2)(3-1)\frac{p}{2} \\ &= 4p - 2. \end{aligned} \tag{25.6.41}$$

$p = 1$（両者が同時に最大限正または負になる場合）では共分散は 2 である。$p = 0$（反転している場合）では共分散は -2 である。最後に、$p = 1/2$（互いに無関係な場合）では共分散は 0 である。したがって、共分散はこれら 2 つの確率変数がどのように関係しているかを測ることがわかる。

共分散についての簡単な注意として、これは線形関係しか測らないという点がある。Y が $\{-2, -1, 0, 1, 2\}$ から等確率で選ばれ、$X = Y^2$ のようなより複雑な関係は見逃されることがある。実際、簡単な計算により、一方が他方の決定論的関数であるにもかかわらず、これらの確率変数の共分散は 0 になる。

連続確率変数についても、話はほぼ同じである。ここまでで離散と連続の移行にはかなり慣れてきたので、導出なしで (25.6.39) の連続版を示す。

$$\sigma_{XY} = \int_{\mathbb{R}^2} (x - \mu_X)(y - \mu_Y) p(x, y) \, dx \, dy. \tag{25.6.42}$$

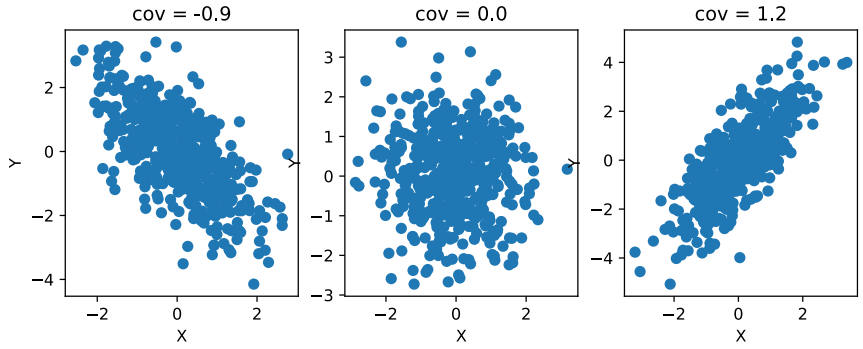
可視化のために、共分散を調整できる確率変数の集まりを見てみよう。

```
# Plot a few random variables adjustable covariance
covs = [-0.9, 0.0, 1.2]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = torch.randn(500)
    Y = covs[i]*X + torch.randn(500)
```

(continues on next page)

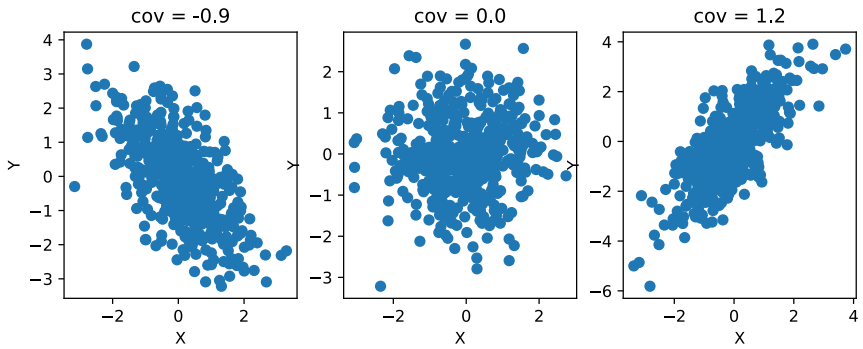
(continued from previous page)

```
    d2l.plt.subplot(1, 4, i+1)
    d2l.plt.scatter(X.numpy(), Y.numpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cov = {covs[i]}')
d2l.plt.show()
```



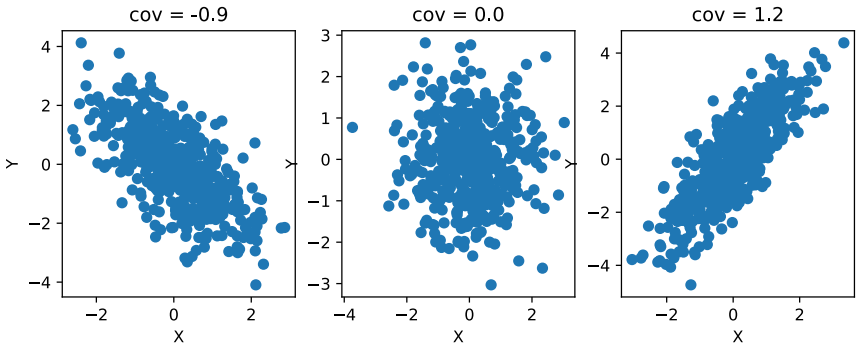

```
# Plot a few random variables adjustable covariance
covs = [-0.9, 0.0, 1.2]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = np.random.normal(0, 1, 500)
    Y = covs[i]*X + np.random.normal(0, 1, (500))

    d2l.plt.subplot(1, 4, i+1)
    d2l.plt.scatter(X.asnumpy(), Y.asnumpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cov = {covs[i]}')
d2l.plt.show()
```



```
# Plot a few random variables adjustable covariance
covs = [-0.9, 0.0, 1.2]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = tf.random.normal((500, ))
    Y = covs[i]*X + tf.random.normal((500, ))

    d2l.plt.subplot(1, 4, i+1)
    d2l.plt.scatter(X.numpy(), Y.numpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cov = {covs[i]}')
d2l.plt.show()
```



共分散の性質をいくつか見てみよう。

- 任意の確率変数 X に対して、$\mathrm{Cov}(X, X) = \mathrm{Var}(X)$。
- 任意の確率変数 X, Y と数 a, b に対して、$\mathrm{Cov}(aX + b, Y) = \mathrm{Cov}(X, aY + b) = a\mathrm{Cov}(X, Y)$。
- X と Y が独立なら、$\mathrm{Cov}(X, Y) = 0$。

さらに、共分散を使うと、先ほど見た関係を拡張できる。X と Y が 2 つの独立な確率変数なら、

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y). \tag{25.6.43}$$

でした。共分散の知識を使うと、この関係を一般化できる。実際、少し代数計算をすると、一般には

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\mathrm{Cov}(X, Y). \tag{25.6.44}$$

となる。これにより、相関した確率変数に対する分散の加法則を一般化できる。

相関

平均と分散の場合と同様に、単位について考えてみよう。X がある単位（たとえばインチ）で測られ、Y が別の単位（たとえばドル）で測られるなら、共分散の単位はこれら 2 つの単位の積 inches × dollars になる。この単位は解釈しにくいことがある。この場合にしばしば欲しいのは、単位に依存しない関係の尺度である。実際、厳密な定量的相関よりも、相関が同じ方向かどうか、そして関係がどれだけ強いかを知りたいことが多いのである。

何が自然かを見るために、思考実験をしてみよう。インチとドルで測っていた確率変数を、インチとセントで測るように変換するとする。この場合、確率変数 Y は 100 倍される。定義をたどると、$\mathrm{Cov}(X,Y)$ も 100 倍されることがわかる。つまり、単位を変えると共分散は 100 倍変わる。したがって、単位不変な相関の尺度を得るには、同じく 100 倍にスケールされる別の量で割る必要がある。そこで有力な候補が標準偏差である。実際、相関係数を

$$\rho(X,Y) = \frac{\mathrm{Cov}(X,Y)}{\sigma_X \sigma_Y}, \tag{25.6.45}$$

と定義すると、これは単位のない値になる。少し数学を使えば、この数は –1 から 1 の間にあり、1 は最大の正の相関、–1 は最大の負の相関を意味することが示せる。

先ほどの具体的な離散例に戻ると、$\sigma_X = 1$、$\sigma_Y = 2$ なので、(25.6.45) を使って 2 つの確率変数の相関を計算すると

$$\rho(X,Y) = \frac{4p - 2}{1 \cdot 2} = 2p - 1. \tag{25.6.46}$$

となる。これは –1 から 1 の範囲をとり、1 が最も相関が強く、–1 が最も相関が弱いという期待通りの振る舞いをする。

別の例として、X を任意の確率変数、$Y = aX + b$ を X の任意の線形決定論的関数とする。このとき、

$$\sigma_Y = \sigma_{aX+b} = |a|\sigma_X, \tag{25.6.47}$$

$$\mathrm{Cov}(X,Y) = \mathrm{Cov}(X, aX + b) = a\mathrm{Cov}(X,X) = a\mathrm{Var}(X), \tag{25.6.48}$$

であり、したがって (25.6.45) より

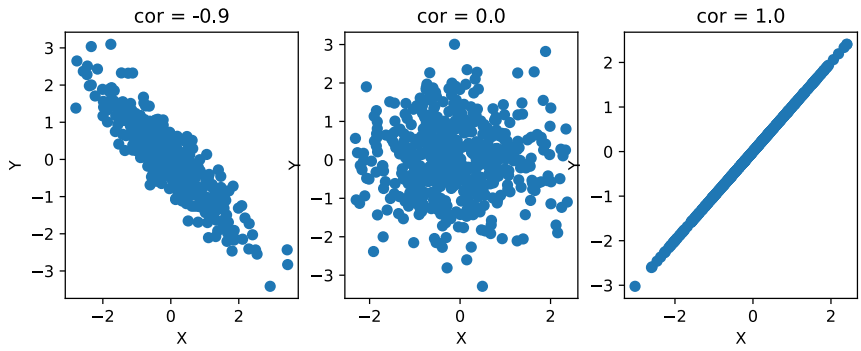
$$\rho(X,Y) = \frac{a\mathrm{Var}(X)}{|a|\sigma_X^2} = \frac{a}{|a|} = \mathrm{sign}(a). \tag{25.6.49}$$

となる。したがって、$a > 0$ なら相関は +1、$a < 0$ なら –1 である。これは、相関が 2 つの確率変数の関係の程度と方向を測るのであって、変動のスケールを測るのではないことを示している。

再び、調整可能な相関をもつ確率変数の集まりを描いてみよう。

```
# Plot a few random variables adjustable correlations
cors = [-0.9, 0.0, 1.0]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = torch.randn(500)
    Y = cors[i] * X + torch.sqrt(torch.tensor(1) -
                                 cors[i]**2) * torch.randn(500)

    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.scatter(X.numpy(), Y.numpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cor = {cors[i]}')
d2l.plt.show()
```



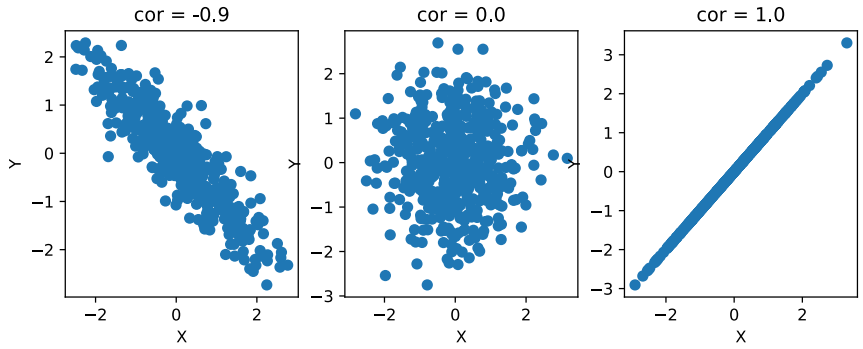

```
# Plot a few random variables adjustable correlations
cors = [-0.9, 0.0, 1.0]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = np.random.normal(0, 1, 500)
    Y = cors[i] * X + np.sqrt(1 - cors[i]**2) * np.random.normal(0, 1, 500)

    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.scatter(X.asnumpy(), Y.asnumpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cor = {cors[i]}')
d2l.plt.show()
```

```
# Plot a few random variables adjustable correlations
cors = [-0.9, 0.0, 1.0]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = tf.random.normal((500, ))
    Y = cors[i] * X + tf.sqrt(tf.constant(1.) -
                                 cors[i]**2) * tf.random.normal((500, ))

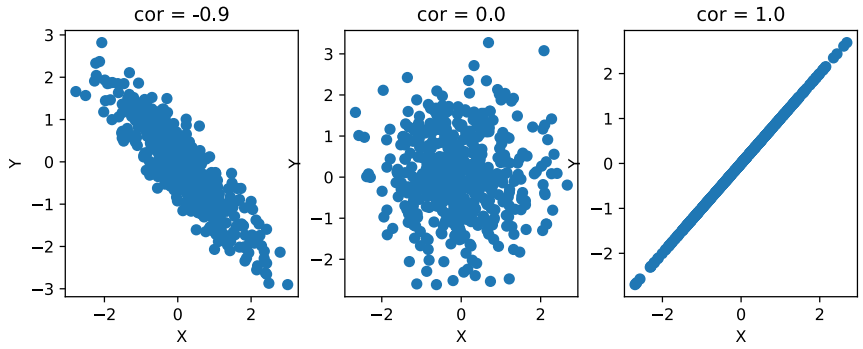
    d2l.plt.subplot(1, 4, i + 1)
```

(continues on next page)



(continued from previous page)

```
    d2l.plt.scatter(X.numpy(), Y.numpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cor = {cors[i]}')
d2l.plt.show()
```



相関の性質をいくつか挙げておきる。

- 任意の確率変数 X に対して、$\rho(X, X) = 1$。
- 任意の確率変数 X, Y と数 a, b に対して、$\rho(aX+b, Y) = \rho(X, aY+b) = \rho(X, Y)$。
- X と Y が独立で、かつ分散が 0 でないなら、$\rho(X, Y) = 0$。

最後に、いくつかの式が見覚えがあるように感じるかもしれない。実際、$\mu_X = \mu_Y = 0$ と仮定してすべて展開すると、これは

$$\rho(X, Y) = \frac{\sum_{i,j} x_i y_i p_{ij}}{\sqrt{\sum_{i,j} x_i^2 p_{ij}} \sqrt{\sum_{i,j} y_j^2 p_{ij}}}. \tag{25.6.50}$$

となる。これは、項の積の和を、項の和の平方根で割った形である。これは、確率 p_{ij} で各座標が重み付けされた 2 つのベクトル $\mathbf{v}, \mathbf{w}$ のなす角の余弦の公式

そのものである。

$$\cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|} = \frac{\sum_i v_i w_i}{\sqrt{\sum_i v_i^2}\sqrt{\sum_i w_i^2}}. \tag{25.6.51}$$

実際、ノルムを標準偏差に、相関を角度の余弦に対応するものと考えれば、幾何学から得られる直感の多くを確率変数の理解に応用できる。

25.6.2 まとめ

- 連続確率変数は、連続体の値をとりうる確率変数である。離散確率変数に比べて扱いが難しい技術的問題がある。
- 確率密度関数を使うと、ある区間で曲線の下の面積がその区間に標本点が入る確率を与える関数として、連続確率変数を扱える。
- 累積分布関数は、確率変数があるしきい値以下である確率である。離散変数と連続変数を統一的に扱える有用な別視点を与える。
- 平均は確率変数の平均値である。
- 分散は、確率変数とその平均の差の二乗の期待値である。
- 標準偏差は分散の平方根である。確率変数がとりうる値の範囲を測るものと考えられる。
- チェビシェフの不等式により、確率変数が大抵の場合に含まれる明示的な区間を与えることで、この直感を厳密にできる。
- 同時密度を使うと、相関した確率変数を扱える。不要な確率変数について積分することで同時密度を周辺化し、目的の確率変数の分布を得られる。
- 共分散と相関係数は、2 つの相関した確率変数の線形関係を測る方法を与える。

25.6.3 演習

1. 密度が $p(x) = \frac{1}{x^2}$ for $x \geq 1$ であり、それ以外では $p(x) = 0$ である確率変数を考える。$P(X > 2)$ はいくらか。
2. ラプラス分布は、密度が $p(x = \frac{1}{2}e^{-|x|}$ で与えられる確率変数である。この関数の平均と標準偏差はいくらか。ヒントとして、$\int_0^\infty xe^{-x}\ dx = 1$ と $\int_0^\infty x^2e^{-x}\ dx = 2$ を用いてよい。
3. 私が街であなたに近づいてきて、「平均が 1、標準偏差が 2 の確率変数を持っていて、標本の 25% が 9 より大きい値をとるのを観測した」と言ったとする。あなたは私の話を信じるだろうか。なぜそう思い、なぜそう思わないのだろうか。
4. 2 つの確率変数 X, Y があり、同時密度が $p_{XY}(x, y) = 4xy$ for $x, y \in [0, 1]$、それ以外では $p_{XY}(x, y) = 0$ で与えられているとする。X と Y の共分散はいくらか。

25.7 最尤推定

機械学習で最もよく出会う考え方の一つが、最尤の観点である。これは、未知のパラメータをもつ確率モデルを扱うとき、データに最も高い確率を与えるパラメータが最もありそうだ、という考え方である。

25.7.1 最尤原理

これはベイズ的な解釈を持っており、考えるうえで役に立ちる。パラメータ $\boldsymbol{\theta}$ をもつモデルと、データ例の集合 X があるとしよう。具体的には、$\boldsymbol{\theta}$ を、コインを投げたときに表が出る確率を表す単一の値、X を独立なコイン投げの列だと考えることができる。この例は後で詳しく見る。

モデルのパラメータとして最もありそうな値を見つけたいなら、次を求めたいことになる。

$$\operatorname{argmax} P(\boldsymbol{\theta} \mid X). : label : eq_max_likе \tag{25.7.1}$$

ベイズの定理によれば、これは次と同じである。

$$\operatorname{argmax} \frac{P(X \mid \boldsymbol{\theta})P(\boldsymbol{\theta})}{P(X)}. \tag{25.7.2}$$

式 $P(X)$ は、データを生成する確率をパラメータに依存せずに表したものなので、$\boldsymbol{\theta}$ にはまったく依存しない。したがって、$\boldsymbol{\theta}$ の最良の選択を変えずにこの項は無視できる。同様に、どのパラメータ集合が他より優れているという事前仮定はないと考えてよいので、$P(\boldsymbol{\theta})$ も θ に依存しないと宣言してよいだろう！たとえば、コイン投げの例では、表が出る確率は $[0, 1]$ の任意の値を取りうるので、それが公平かどうかについて事前の信念はない（これはしばしば 無情報事前分布と呼ばれる）。したがって、ベイズの定理の適用により、$\boldsymbol{\theta}$ の最良の選択は $\boldsymbol{\theta}$ の最尤推定量であることがわかる。

$$\hat{\boldsymbol{\theta}} = \operatorname*{argmax}_{\boldsymbol{\theta}} P(X \mid \boldsymbol{\theta}). \tag{25.7.3}$$

一般的な用語として、パラメータが与えられたときのデータの確率（$P(X \mid \boldsymbol{\theta})$）は 尤度と呼ばれる。

具体例

これが具体的にどう働くかを見てみよう。コイン投げで表が出る確率を表す単一のパラメータ θ があるとする。このとき、裏が出る確率は $1 - \theta$ である。観測データ X が n_H 回の表と n_T 回の裏からなる列であれば、独立な確率は掛け算できるので、

$$P(X \mid \theta) = \theta^{n_H}(1 - \theta)^{n_T}. \tag{25.7.4}$$

13 枚のコインを投げて“HHHTHTTHHHHHT”という列が得られたとする。これは $n_H = 9$、$n_T = 4$ だろうから、

$$P(X \mid \theta) = \theta^9(1-\theta)^4. \tag{25.7.5}$$

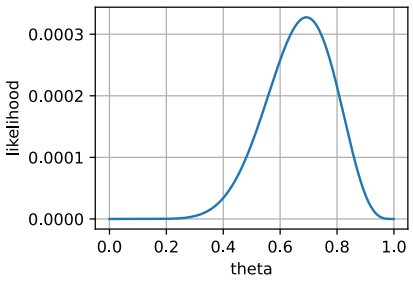
この例のよい点の一つは、答えをあらかじめ知っていることである。実際、口頭で「13 回コインを投げて、9 回表が出た。では、そのコインが表を出す確率の最良の推定値は？」と言えば、誰もが正しく 9/13 と答えるだろう。この最尤法が与えてくれるのは、その数を第一原理から求める方法であり、しかもそれははるかに複雑な状況へ一般化できる。

この例に対する $P(X \mid \theta)$ のグラフは次のとおりである。

```
%matplotlib inline
from d2l import torch as d2l
import torch

theta = torch.arange(0, 1, 0.001)
p = theta**9 * (1 - theta)**4.

d2l.plot(theta, p, 'theta', 'likelihood')
```




```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, np, npx
npx.set_np()

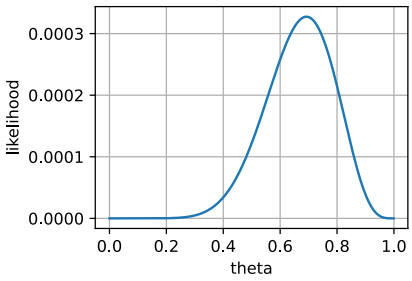
theta = np.arange(0, 1, 0.001)
p = theta**9 * (1 - theta)**4.

d2l.plot(theta, p, 'theta', 'likelihood')
```

```
[06:59:41] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import tensorflow as tf
```

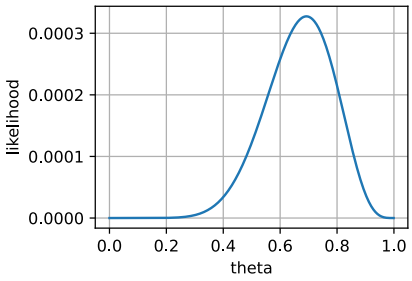
(continues on next page)



(continued from previous page)

```
theta = tf.range(0, 1, 0.001)
p = theta**9 * (1 - theta)**4.

d2l.plot(theta, p, 'theta', 'likelihood')
```



これは、期待される $9/13 \approx 0.7\ldots$ の近くで最大値をとりる。これが正確にそこなのかを見るには、微積分を使える。最大値では関数の傾きが 0 になることに注意しよう。したがって、導関数が 0 となる θ の値を見つけ、その中で最も高い確率を与えるものを選べば、最尤推定 `eq_max_like` を求められる。計算すると、

$$\begin{aligned} 0 &= \frac{d}{d\theta} P(X \mid \theta) \\ &= \frac{d}{d\theta} \theta^9 (1-\theta)^4 \\ &= 9\theta^8 (1-\theta)^4 - 4\theta^9 (1-\theta)^3 \\ &= \theta^8 (1-\theta)^3 (9 - 13\theta). \end{aligned} \tag{25.7.6}$$

これには 3 つの解、0、1、$9/13$ がある。最初の 2 つは、列に確率 0 を割り当てるので、明らかに最大値ではなく最小値である。最後の値は列に 0 の確率を割り当てないので、最尤推定量 $\hat{\theta} = 9/13$ でなければならない。

25.7.2 数値最適化と負の対数尤度

前の例はよいものであるが、もし何十億ものパラメータやデータ例があったらどうだろうか？

まず、すべてのデータ例が独立であると仮定すると、尤度は多数の確率の積になるため、実用上そのまま扱うことはできない。実際、各確率は $[0, 1]$ にあり、典型的にはおよそ $1/2$ 程度の値だろうから、$(1/2)^{1000000000}$ の積は機械精度をはるかに下回りる。これを直接扱うことはできない。

しかし、対数をとると積が和に変わることを思い出する。このとき

$$\log((1/2)^{1000000000}) = 1000000000 \cdot \log(1/2) \approx -301029995.6\ldots \tag{25.7.7}$$

となる。この数は、単精度の 32 ビット浮動小数点数にすら十分収まりる。したがって、対数尤度を考えるべきである。これは

$$\log(P(X \mid \boldsymbol{\theta})). \tag{25.7.8}$$

である。関数 $x \mapsto \log(x)$ は単調増加なので、尤度を最大化することは対数尤度を最大化することと同じである。実際、Section 25.9 では、ナイーブベイズ分類器という具体例でこの考え方が使われるのを見る。

私たちはしばしば損失関数を扱い、その損失を最小化したいと考える。最大尤度は、$-\log(P(X \mid \boldsymbol{\theta}))$ をとることで損失の最小化に変えられる。これが 負の対数尤度である。

これを示すために、先ほどのコイン投げ問題を考え、閉形式解を知らないふりをしよう。すると、

$$-\log(P(X \mid \boldsymbol{\theta})) = -\log(\theta^{n_H}(1-\theta)^{n_T}) = -(n_H \log(\theta) + n_T \log(1-\theta)). \tag{25.7.9}$$

と計算できる。これはコードに書けますし、何十億回のコイン投げに対しても自由に最適化できる。

```
# Set up our data
n_H = 8675309
n_T = 256245

# Initialize our paramteres
theta = torch.tensor(0.5, requires_grad=True)

# Perform gradient descent
lr = 1e-9
for iter in range(100):
    loss = -(n_H * torch.log(theta) + n_T * torch.log(1 - theta))
    loss.backward()
    with torch.no_grad():
        theta -= lr * theta.grad
    theta.grad.zero_()

# Check output
theta, n_H / (n_H + n_T)
```

```
(tensor(0.9713, requires_grad=True), 0.9713101437890875)
```

```
# Set up our data
n_H = 8675309
n_T = 256245

# Initialize our paramteres
theta = np.array(0.5)
theta.attach_grad()

# Perform gradient descent
lr = 1e-9
for iter in range(100):
    with autograd.record():
        loss = -(n_H * np.log(theta) + n_T * np.log(1 - theta))
    loss.backward()
    theta -= lr * theta.grad

# Check output
theta, n_H / (n_H + n_T)
```

```
[06:59:41] ../src/base.cc:48: GPU context requested, but no GPUs found.
```

```
(array(0.9713101), 0.9713101437890875)
```

```
# Set up our data
n_H = 8675309
n_T = 256245

# Initialize our paramteres
theta = tf.Variable(tf.constant(0.5))

# Perform gradient descent
lr = 1e-9
for iter in range(100):
    with tf.GradientTape() as t:
        loss = -(n_H * tf.math.log(theta) + n_T * tf.math.log(1 - theta))
    theta.assign_sub(lr * t.gradient(loss, theta))

# Check output
theta, n_H / (n_H + n_T)
```

```
(<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.9713101>,
 0.9713101437890875)
```

数値計算上の都合だけが、負の対数尤度を使う理由ではない。ほかにも、これを好む理由がいくつかある。

対数尤度を考える第 2 の理由は、微積分の規則を簡単に適用できることである。上で述べたように、独立性の仮定により、機械学習で出会う確率の多くは

個々の確率の積になる。

$$P(X \mid \boldsymbol{\theta}) = p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}). \tag{25.7.10}$$

これは、導関数を求めるために直接積の微分法則を適用すると、

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{\theta}} P(X \mid \boldsymbol{\theta}) = &\left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_1 \mid \boldsymbol{\theta})\right) \cdot P(x_2 \mid \boldsymbol{\theta}) \cdots P(x_n \mid \boldsymbol{\theta}) \\ &+ P(x_1 \mid \boldsymbol{\theta}) \cdot \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_2 \mid \boldsymbol{\theta})\right) \cdots P(x_n \mid \boldsymbol{\theta}) \\ &\vdots \\ &+ P(x_1 \mid \boldsymbol{\theta}) \cdot P(x_2 \mid \boldsymbol{\theta}) \cdots \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_n \mid \boldsymbol{\theta})\right).\end{aligned} \tag{25.7.11}$$

となる。これには $n(n-1)$ 回の乗算と $(n-1)$ 回の加算が必要なので、入力に対して二次時間に比例する！項のまとめ方をうまく工夫すれば線形時間に減らせるが、少し考える必要がある。これに対して負の対数尤度では、

$$-\log\left(P(X \mid \boldsymbol{\theta})\right) = -\log(P(x_1 \mid \boldsymbol{\theta})) - \log(P(x_2 \mid \boldsymbol{\theta})) \cdots - \log(P(x_n \mid \boldsymbol{\theta})), \tag{25.7.12}$$

となり、したがって

$$-\frac{\partial}{\partial \boldsymbol{\theta}} \log\left(P(X \mid \boldsymbol{\theta})\right) = \frac{1}{P(x_1 \mid \boldsymbol{\theta})} \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_1 \mid \boldsymbol{\theta})\right) + \cdots + \frac{1}{P(x_n \mid \boldsymbol{\theta})} \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_n \mid \boldsymbol{\theta})\right). \tag{25.7.13}$$

これには n 回の除算と $n-1$ 回の和しか必要なく、入力に対して線形時間である。

負の対数尤度を考える第 3 の、そして最後の理由は、情報理論との関係である。これについては Section 25.11 で詳しく議論する。これは、確率変数における情報量やランダムさの程度を測る方法を与える厳密な数学理論である。この分野で中心となる対象はエントロピーであり、

$$H(p) = -\sum_i p_i \log_2(p_i), \tag{25.7.14}$$

である。これはソースのランダムさを測りる。これが単に平均の $-\log$ 確率にほかならないことに注意する。したがって、負の対数尤度をデータ例の数で割れば、クロスエントロピーとして知られるエントロピーの親戚が得られる。この理論的解釈だけでも、データセット全体にわたる平均負の対数尤度をモデル性能の指標として報告する十分な動機になる。

25.7.3 連続変数に対する最尤推定

これまで行ってきたことはすべて離散確率変数を扱うことを前提としていたが、連続変数を扱いたい場合はどうだろうか？

短く言えば、確率をすべて確率密度に置き換える以外、何も変わりない。密度

は小文字の p で書くことを思い出すと、たとえば今は次のように書く。

$$-\log\left(p(X \mid \boldsymbol{\theta})\right) = -\log(p(x_1 \mid \boldsymbol{\theta})) - \log(p(x_2 \mid \boldsymbol{\theta})) \cdots - \log(p(x_n \mid \boldsymbol{\theta})) = -\sum_i \log(p(x_i \mid \theta)). \tag{25.7.15}$$

問題は、「なぜこれでよいのか？」である。そもそも、密度を導入した理由は、特定の結果が起こる確率そのものが 0 だからでした。では、どのパラメータ集合に対してもデータを生成する確率は 0 ではないのだろうか？

実際そのとおりである。そして、なぜ密度に切り替えられるのかを理解するには、ϵ に何が起こるかを追跡する練習になる。

まず、目的を再定義しよう。連続確率変数については、ちょうど正しい値を得る確率を計算するのではなく、ある範囲 ϵ 以内で一致することを考えるとする。簡単のため、データは同一分布の確率変数 $X_1, \ldots, X_N$ の繰り返し観測 $x_1, \ldots, x_N$ だと仮定する。前に見たように、これは次のように書ける。

$$\begin{aligned} &P(X_1 \in [x_1, x_1 + \epsilon], X_2 \in [x_2, x_2 + \epsilon], \ldots, X_N \in [x_N, x_N + \epsilon] \mid \boldsymbol{\theta}) \\ \approx &\epsilon^N p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}). \end{aligned} \tag{25.7.16}$$

したがって、これの負の対数をとると、

$$\begin{aligned} &-\log(P(X_1 \in [x_1, x_1 + \epsilon], X_2 \in [x_2, x_2 + \epsilon], \ldots, X_N \in [x_N, x_N + \epsilon] \mid \boldsymbol{\theta})) \\ \approx &-N\log(\epsilon) - \sum_i \log(p(x_i \mid \boldsymbol{\theta})). \end{aligned} \tag{25.7.17}$$

となる。この式を見ると、ϵ が現れるのは加法定数 $-N\log(\epsilon)$ の部分だけである。これは $\boldsymbol{\theta}$ にまったく依存しないので、$\boldsymbol{\theta}$ の最適な選択は ϵ の選び方に依存しない！4 桁の精度を要求しても 400 桁を要求しても、$\boldsymbol{\theta}$ の最良の選択は同じである。したがって、ϵ を自由に捨てて、最適化したいものが

$$-\sum_i \log(p(x_i \mid \boldsymbol{\theta})). \tag{25.7.18}$$

であることがわかる。

このように、最大尤度の観点は、確率を確率密度に置き換えるだけで、離散確率変数と同じくらい容易に連続確率変数にも適用できることがわかる。

25.7.4 まとめ

- 最尤原理は、与えられたデータセットに最もよく適合するモデルとは、そのデータを最も高い確率で生成するモデルだと述べる。
- 実際には、数値安定性、積を和に変換できること（およびそれに伴う勾配計算の簡略化）、情報理論との理論的なつながりなど、さまざまな理由から負の対数尤度がよく使われる。
- 離散設定で最も説明しやすいであるが、データ点に割り当てられた確率密度を最大化することで、連続設定にも自由に一般化できる。

25.7.5 演習

1. 非負の確率変数が、ある $\alpha > 0$ に対して密度 $\alpha e^{-\alpha x}$ をもつことがわかっているとする。この確率変数から 1 つの観測値として数 3 を得た。α の最尤推定値は何だろうか？
2. 平均は未知だが分散は 1 のガウス分布から、サンプル $\{x_i\}_{i=1}^N$ からなるデータセットが与えられているとする。平均の最尤推定値は何だろうか？

25.8 分布

これまで、離散的な場合と連続的な場合の両方で確率を扱う方法を学んできた。ここでは、よく出会う代表的な分布について見ていきよう。機械学習の分野によっては、これらよりはるかに多くの分布に精通している必要があるかもしれないし、深層学習のある分野ではまったく不要なこともある。しかし、基本的な一覧として知っておくのは有益である。まず、いくつかの共通ライブラリをインポートしよう。

```
%matplotlib inline
from d2l import torch as d2l
from IPython import display
from math import erf, factorial
import torch

torch.pi = torch.acos(torch.zeros(1)) * 2  # Define pi in torch
```

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from math import erf, factorial
import numpy as np
```

```
p = 0.3

d2l.set_figsize()
d2l.plt.stem([0, 1], [1 - p, p], use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```

```
%matplotlib inline
from d2l import tensorflow as d2l
from IPython import display
from math import erf, factorial
import tensorflow as tf
```

(continues on next page)

(continued from previous page)

```
import tensorflow_probability as tfp

tf.pi = tf.acos(tf.zeros(1)) * 2  # Define pi in TensorFlow
```

25.8.1 ベルヌーイ分布

これは通常最初に出会う最も単純な確率変数である。この確率変数は、コイン投げを表し、1 が確率 p、0 が確率 $1-p$ で出ることを表する。この分布に従う確率変数 X は

$$X \sim \text{Bernoulli}(p). \tag{25.8.1}$$

と書く。

累積分布関数は

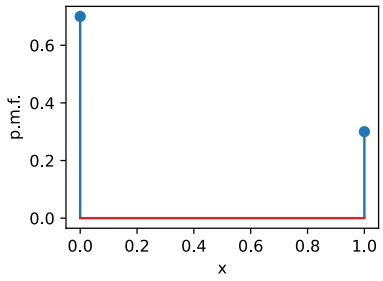
$$F(x) = \begin{cases} 0 & x < 0, \\ 1-p & 0 \le x < 1, \\ 1 & x >= 1. \end{cases} \tag{25.8.2}$$

である。

確率質量関数を以下に描く。

```
p = 0.3

d2l.set_figsize()
d2l.plt.stem([0, 1], [1 - p, p], use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```



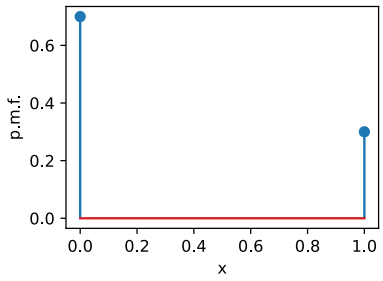

```
p = 0.3

d2l.set_figsize()
d2l.plt.stem([0, 1], [1 - p, p], use_line_collection=True)
d2l.plt.xlabel('x')
```

(continues on next page)

(continued from previous page)

```
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```

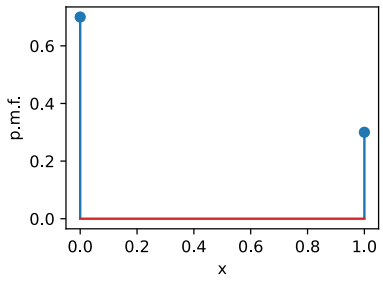



```
n = 5

d2l.plt.stem([i+1 for i in range(n)], n*[1 / n], use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```

```
p = 0.3

d2l.set_figsize()
d2l.plt.stem([0, 1], [1 - p, p], use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```

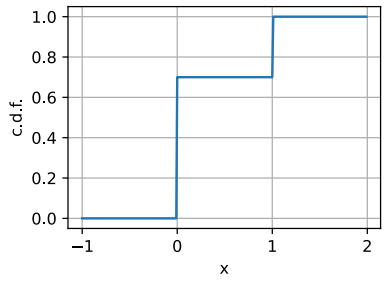


では、累積分布関数 (25.8.2) を描いてみよう。

```
x = torch.arange(-1, 2, 0.01)

def F(x):
    return 0 if x < 0 else 1 if x > 1 else 1 - p

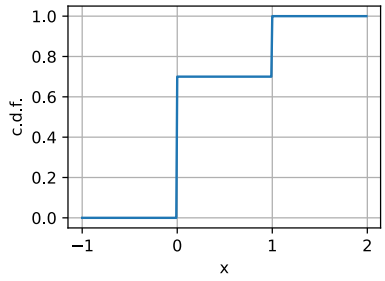
d2l.plot(x, torch.tensor([F(y) for y in x]), 'x', 'c.d.f.')
```




```
x = np.arange(-1, 2, 0.01)

def F(x):
    return 0 if x < 0 else 1 if x > 1 else 1 - p

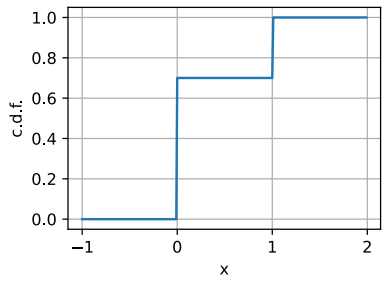
d2l.plot(x, np.array([F(y) for y in x]), 'x', 'c.d.f.')
```




```
x = tf.range(-1, 2, 0.01)

def F(x):
    return 0 if x < 0 else 1 if x > 1 else 1 - p

d2l.plot(x, tf.constant([F(y) for y in x]), 'x', 'c.d.f.')
```



$X \sim \text{Bernoulli}(p)$ ならば、

- $\mu_X = p$、

- $\sigma_X^2 = p(1-p)$。

ベルヌーイ確率変数から任意の形状の配列をサンプルするには、次のようにする。

```
1*(torch.rand(10, 10) < p)
```

```
tensor([[0, 0, 0, 1, 0, 0, 1, 0, 0, 1],
        [0, 1, 1, 1, 0, 0, 1, 1, 1, 0],
        [0, 0, 1, 0, 0, 0, 0, 1, 0, 1],
        [0, 0, 0, 1, 0, 1, 0, 0, 0, 1],
        [0, 1, 1, 0, 1, 1, 1, 1, 0, 1],
        [1, 0, 0, 0, 0, 0, 1, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 1, 0, 1, 0, 0, 0, 1],
        [0, 1, 0, 0, 1, 0, 0, 0, 0, 0]])
```

```
1*(np.random.rand(10, 10) < p)
```

```
array([[0, 0, 1, 0, 0, 0, 1, 0, 0, 1],
       [1, 0, 0, 1, 0, 0, 1, 0, 0, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [1, 0, 1, 1, 0, 0, 0, 0, 0, 0],
       [1, 1, 0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 1, 0, 1, 1, 0, 1, 0, 0],
       [1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0, 0, 0, 0]])
```

```
tf.cast(tf.random.uniform((10, 10)) < p, dtype=tf.float32)
```

```
<tf.Tensor: shape=(10, 10), dtype=float32, numpy=
array([[0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 1., 0., 1., 1., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 1., 1., 0., 0., 0.],
       [1., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 1., 1., 0., 0., 1.],
       [0., 0., 0., 1., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 1., 0., 0., 0.]], dtype=float32)>
```

25.8.2 離散一様分布

次によく現れる確率変数は離散一様分布である。ここでは、$\{1, 2, \ldots, n\}$ の整数上に台を持つものを考えるが、他の任意の値集合を自由に選んでも構いない。

この文脈で 一様というのは、取りうるすべての値が等確率であることを意味する。各値 $i \in \{1, 2, 3, \ldots, n\}$ の確率は $p_i = \frac{1}{n}$ である。この分布に従う確率変数 X は

$$X \sim U(n). \tag{25.8.3}$$

と表する。

累積分布関数は

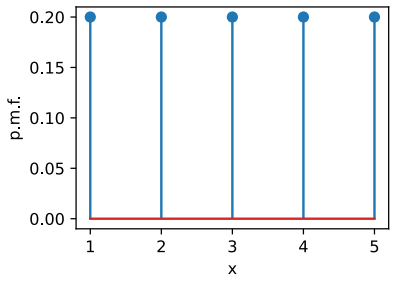
$$F(x) = \begin{cases} 0 & x < 1, \\ \frac{k}{n} & k \leq x < k+1 \text{ with } 1 \leq k < n, \\ 1 & x >= n. \end{cases} \tag{25.8.4}$$

である。

まず確率質量関数を描きよう。

```
n = 5

d2l.plt.stem([i+1 for i in range(n)], n*[1 / n], use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```



では、累積分布関数 (25.8.4) を描いてみよう。

```
x = torch.arange(-1, 6, 0.01)

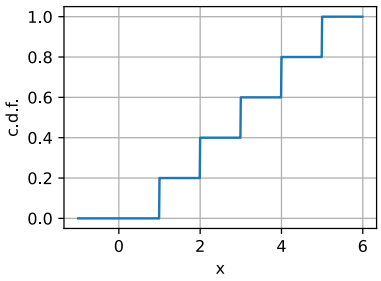
def F(x):
    return 0 if x < 1 else 1 if x > n else torch.floor(x) / n

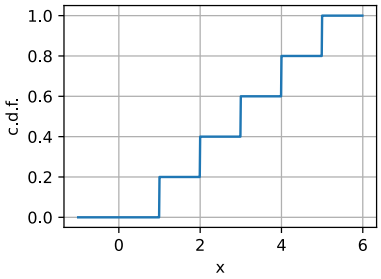
d2l.plot(x, torch.tensor([F(y) for y in x]), 'x', 'c.d.f.')
```

```
x = np.arange(-1, 6, 0.01)

def F(x):
    return 0 if x < 1 else 1 if x > n else np.floor(x) / n

d2l.plot(x, np.array([F(y) for y in x]), 'x', 'c.d.f.')
```

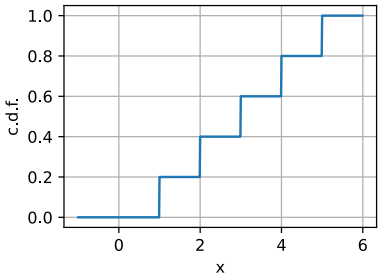





```
x = tf.range(-1, 6, 0.01)

def F(x):
    return 0 if x < 1 else 1 if x > n else tf.floor(x) / n

d2l.plot(x, [F(y) for y in x], 'x', 'c.d.f.')
```



$X \sim U(n)$ ならば、

- $\mu_X = \frac{1+n}{2}$、
- $\sigma_X^2 = \frac{n^2-1}{12}$。

離散一様確率変数から任意の形状の配列をサンプルするには、次のようにする。

```
torch.randint(1, n, size=(10, 10))
```

```
tensor([[3, 3, 4, 2, 1, 4, 3, 3, 3, 1],
        [2, 3, 4, 4, 1, 1, 4, 3, 4, 2],
        [1, 2, 1, 2, 2, 3, 2, 4, 3, 3],
        [3, 4, 1, 4, 2, 3, 1, 2, 3, 3],
        [1, 2, 4, 1, 2, 4, 1, 3, 2, 1],
        [3, 2, 2, 1, 2, 4, 2, 4, 2, 1],
        [4, 2, 2, 1, 3, 4, 3, 3, 1, 4],
        [4, 4, 4, 1, 1, 3, 3, 2, 3, 4],
        [4, 3, 3, 3, 4, 4, 4, 2, 2, 4],
        [2, 2, 2, 2, 1, 2, 2, 1, 4, 2]])
```

```
np.random.randint(1, n, size=(10, 10))
```

```
array([[2, 2, 2, 3, 4, 1, 4, 2, 1, 2],
       [1, 1, 3, 3, 4, 1, 2, 4, 3, 1],
       [4, 2, 1, 3, 2, 2, 1, 4, 1, 2],
       [3, 4, 1, 1, 2, 4, 2, 4, 4, 1],
       [1, 2, 2, 1, 4, 1, 1, 1, 4, 4],
       [4, 4, 2, 3, 1, 2, 2, 2, 2, 1],
       [2, 3, 1, 4, 2, 3, 2, 4, 4, 3],
       [4, 1, 3, 1, 3, 2, 2, 4, 2, 1],
       [4, 3, 2, 4, 4, 3, 1, 1, 1, 2],
       [2, 2, 1, 4, 1, 2, 4, 1, 1, 1]])
```

```
tf.random.uniform((10, 10), 1, n, dtype=tf.int32)
```

```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[4, 1, 3, 1, 4, 2, 3, 3, 3, 2],
       [4, 3, 1, 2, 4, 4, 3, 1, 4, 4],
       [4, 1, 4, 3, 3, 1, 4, 3, 4, 4],
       [4, 2, 2, 2, 3, 3, 1, 3, 1, 4],
       [3, 1, 3, 3, 1, 3, 4, 4, 3, 2],
       [1, 1, 4, 1, 3, 1, 3, 4, 3, 1],
       [1, 4, 4, 1, 3, 4, 4, 2, 3, 4],
       [4, 3, 2, 3, 2, 2, 4, 1, 3, 1],
       [1, 3, 3, 1, 3, 4, 3, 4, 2, 3],
       [4, 2, 2, 2, 1, 3, 2, 2, 1, 3]], dtype=int32)>
```

25.8.3 連続一様分布

次に、連続一様分布について説明しよう。この確率変数の考え方は、離散一様分布の n を増やし、それを区間 $[a, b]$ に収まるようにスケーリングすると、$[a, b]$ の中の任意の値をすべて等確率で選ぶ連続確率変数に近づく、というものである。この分布は

$$X \sim U(a, b). \tag{25.8.5}$$

と表する。

確率密度関数は

$$p(x) = \begin{cases} \frac{1}{b-a} & x \in [a, b], \\ 0 & x \notin [a, b]. \end{cases} \tag{25.8.6}$$

である。

累積分布関数は

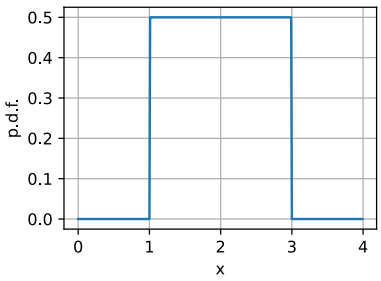
$$F(x) = \begin{cases} 0 & x < a, \\ \frac{x-a}{b-a} & x \in [a, b], \\ 1 & x >= b. \end{cases} \tag{25.8.7}$$

である。

まず確率密度関数 (25.8.6) を描きよう。

```
a, b = 1, 3

x = torch.arange(0, 4, 0.01)
p = (x > a).type(torch.float32)*(x < b).type(torch.float32)/(b-a)
d2l.plot(x, p, 'x', 'p.d.f.')
```




```
a, b = 1, 3

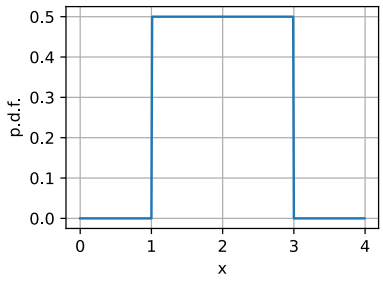
x = np.arange(0, 4, 0.01)
p = (x > a)*(x < b)/(b - a)

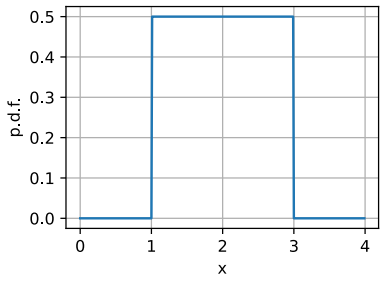
d2l.plot(x, p, 'x', 'p.d.f.')
```

```
a, b = 1, 3

x = tf.range(0, 4, 0.01)
p = tf.cast(x > a, tf.float32) * tf.cast(x < b, tf.float32) / (b - a)
d2l.plot(x, p, 'x', 'p.d.f.')
```

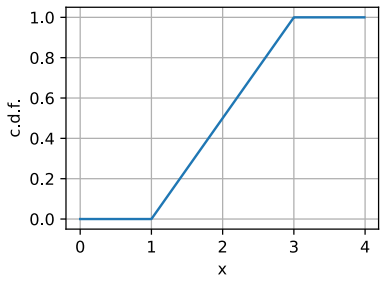
では、累積分布関数 (25.8.7) を描いてみよう。






```
def F(x):
    return 0 if x < a else 1 if x > b else (x - a) / (b - a)

d2l.plot(x, torch.tensor([F(y) for y in x]), 'x', 'c.d.f.')
```



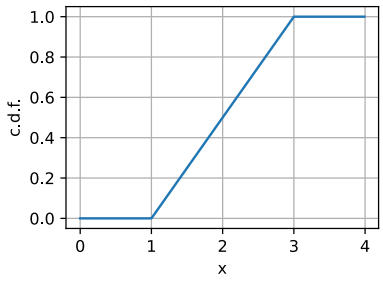

```
def F(x):
    return 0 if x < a else 1 if x > b else (x - a) / (b - a)

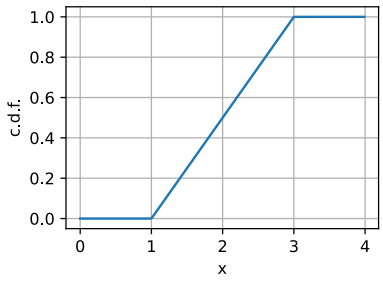
d2l.plot(x, np.array([F(y) for y in x]), 'x', 'c.d.f.')
```

```
def F(x):
    return 0 if x < a else 1 if x > b else (x - a) / (b - a)

d2l.plot(x, [F(y) for y in x], 'x', 'c.d.f.')
```

$X \sim U(a, b)$ ならば、





- $\mu_X = \frac{a+b}{2}$、
- $\sigma_X^2 = \frac{(b-a)^2}{12}$。

一様確率変数から任意の形状の配列をサンプルするには、次のようにする。デフォルトでは $U(0,1)$ からサンプルされるので、別の範囲が欲しい場合はスケーリングする必要がある。

```
(b - a) * torch.rand(10, 10) + a
```

```
tensor([[2.0679, 2.4100, 2.5762, 1.8499, 1.0621, 1.9996, 2.3631, 2.6283, 1.
↪6981,
         1.3658],
        [2.2581, 2.8063, 2.5712, 2.9981, 1.2781, 1.3680, 2.5652, 2.8932, 1.
↪1143,
         1.6694],
        [2.9813, 2.2677, 2.7297, 1.4408, 2.9802, 2.4943, 2.5363, 1.6275, 1.
↪4729,
         2.5822],
        [2.9420, 2.5521, 2.1417, 1.2070, 1.2284, 2.7954, 2.9322, 1.7272, 2.
↪8715,
         1.5982],
        [2.3660, 1.6054, 2.5989, 2.2134, 1.2444, 2.1781, 1.5832, 2.1513, 1.
↪8993,
         2.8935],
        [1.7960, 2.7038, 1.1465, 1.3827, 1.9866, 2.1409, 2.0720, 1.9597, 2.
↪5218,
         2.1478],
        [1.8626, 2.3913, 2.0165, 2.2057, 1.9050, 2.0541, 1.1899, 2.6062, 2.
```

(continues on next page)

(continued from previous page)

```
↪8070,
         1.3668],
        [2.4310, 1.1270, 1.9386, 2.7942, 1.9744, 2.6841, 2.5107, 1.2452, 2.
↪3955,
         1.8620],
        [2.0176, 2.9634, 1.2320, 2.4971, 1.4820, 2.8725, 2.6219, 2.1171, 2.
↪1623,
         1.8533],
        [2.1182, 1.2056, 1.6436, 1.7542, 1.7792, 2.2062, 2.9191, 1.1343, 2.
↪0459,
         1.9995]])
```

```
(b - a) * np.random.rand(10, 10) + a
```

```
array([[1.10493389, 2.18332184, 2.55880868, 2.40644506, 1.98823581,
        2.03304566, 2.54314285, 2.28893216, 1.96832242, 1.21947864],
       [1.00116734, 1.87327363, 2.06307085, 1.37704206, 2.98520983,
        2.85361327, 2.614708  , 1.67971392, 2.31319471, 2.08720314],
       [2.9417875 , 2.58942767, 1.36652719, 2.76520312, 1.6291505 ,
        2.57007544, 2.18738455, 1.08755762, 1.69884709, 1.9310117 ],
       [1.82504201, 1.80920424, 1.52510532, 2.17378476, 1.211283  ,
        2.15483981, 2.14476987, 1.30869007, 2.7514197 , 2.93051749],
       [1.46794909, 1.1367775 , 1.35302173, 2.6377266 , 2.43745649,
        2.98864716, 1.26447563, 1.48047024, 2.08178225, 1.32105626],
       [1.25845706, 2.03397717, 2.73034409, 2.70469505, 2.34734372,
        1.74027474, 2.96683054, 1.37055974, 1.53226889, 2.36267969],
       [2.94902185, 1.77360701, 1.30736801, 2.21610289, 2.60674295,
        1.97389815, 1.96275176, 2.9917009 , 2.66415546, 2.43568684],
       [1.41549567, 2.78340407, 1.49017239, 1.92110928, 2.44749996,
        1.94183467, 2.55685336, 2.06993835, 1.62822726, 1.99831404],
       [2.4445657 , 2.44027101, 2.11419221, 1.94853489, 1.81220577,
        1.74003385, 1.05746194, 1.94546557, 2.53363203, 1.74536491],
       [2.2648276 , 2.99783037, 2.15039751, 1.75704603, 1.46434226,
        2.05093236, 1.80039954, 1.37730793, 2.03797871, 1.62589064]])
```

```
(b - a) * tf.random.uniform((10, 10)) + a
```

```
<tf.Tensor: shape=(10, 10), dtype=float32, numpy=
array([[1.5681243, 1.8772495, 2.4839144, 1.5758138, 1.51053  , 2.5540862,
        1.1616733, 2.970998 , 2.3602724, 1.4549215],
       [1.1370218, 1.2265592, 1.267734 , 2.7446856, 1.3154321, 2.1516864,
        2.9253604, 1.2757335, 1.1784961, 1.4303627],
       [1.3996754, 1.6577711, 2.4583366, 2.9051504, 2.5880024, 1.2249057,
        2.5468373, 1.6416607, 2.4872744, 2.6817133],
       [2.1866   , 2.4692721, 2.0441496, 1.6734836, 2.4291286, 2.7591374,
        2.8040156, 2.3101315, 2.9683428, 2.2392678],
       [1.4663892, 2.7784042, 2.6934195, 2.1465902, 1.9948542, 1.8531013,
        2.4695644, 1.7235866, 1.9122441, 1.3478694],
       [1.7405694, 1.8584762, 1.3478763, 1.194274 , 1.7994637, 2.0353231,
        1.4105015, 2.9599195, 1.3373148, 1.2692168],
```

(continues on next page)

(continued from previous page)

```
       [2.0873346, 2.6769884, 2.5507674, 2.666769 , 2.8097057, 2.0322669,
        1.9368191, 2.791131 , 2.3000367, 1.9854867],
       [1.270735 , 1.7274289, 1.680018 , 1.3198214, 2.4480646, 2.0401797,
        2.9680197, 1.4028776, 1.9465854, 2.1036496],
       [2.446667 , 1.9727476, 2.4787436, 1.896085 , 1.8473601, 2.78446  ,
        1.8548636, 2.0098755, 2.0615952, 1.9896467],
       [1.8838322, 2.0051875, 2.7124982, 2.7813394, 1.4981465, 2.5291965,
        1.2720184, 2.5645955, 1.8139408, 2.5306706]], dtype=float32)>
```

25.8.4 二項分布

少し複雑にして、二項確率変数を見てみよう。この確率変数は、成功確率 p の独立な試行を n 回行い、成功が何回起こるかを数えることから生まれる。

これを数学的に表しよう。各試行は独立な確率変数 X_i であり、成功を 1、失敗を 0 で表する。それぞれが成功確率 p の独立なコイン投げなので、$X_i \sim \text{Bernoulli}(p)$ と書ける。すると、二項確率変数は

$$X = \sum_{i=1}^{n} X_i. \tag{25.8.8}$$

である。

このとき、

$$X \sim \text{Binomial}(n, p). \tag{25.8.9}$$

と書く。

累積分布関数を得るには、ちょうど k 回成功する場合が $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ 通りあり、それぞれが確率 $p^k(1-p)^{n-k}$ で起こることに注意する必要がある。したがって、累積分布関数は

$$F(x) = \begin{cases} 0 & x < 0, \\ \sum_{m \leq k} \binom{n}{m} p^m (1-p)^{n-m} & k \leq x < k+1 \text{ with } 0 \leq k < n, \\ 1 & x >= n. \end{cases} \tag{25.8.10}$$

である。

まず確率質量関数を描きよう。

```
n, p = 10, 0.2

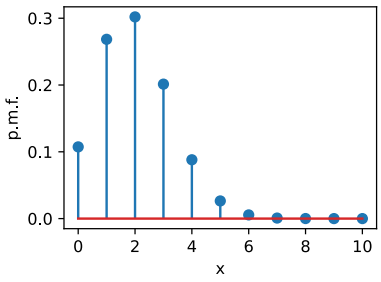
# Compute binomial coefficient
def binom(n, k):
    comb = 1
    for i in range(min(k, n - k)):
        comb = comb * (n - i) // (i + 1)
    return comb
```

(continues on next page)

(continued from previous page)

```
pmf = d2l.tensor([p**i * (1-p)**(n - i) * binom(n, i) for i in range(n + 1)])

d2l.plt.stem([i for i in range(n + 1)], pmf, use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```

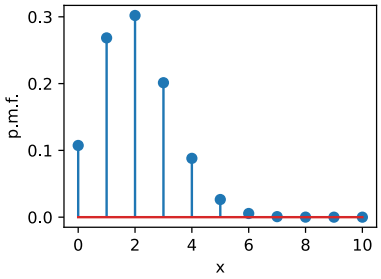



```
n, p = 10, 0.2

# Compute binomial coefficient
def binom(n, k):
    comb = 1
    for i in range(min(k, n - k)):
        comb = comb * (n - i) // (i + 1)
    return comb

pmf = np.array([p**i * (1-p)**(n - i) * binom(n, i) for i in range(n + 1)])

d2l.plt.stem([i for i in range(n + 1)], pmf, use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```




```
n, p = 10, 0.2

# Compute binomial coefficient
def binom(n, k):
```

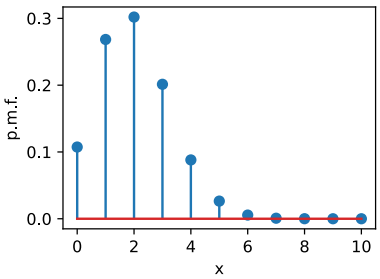
(continues on next page)

(continued from previous page)

```
    comb = 1
    for i in range(min(k, n - k)):
        comb = comb * (n - i) // (i + 1)
    return comb

pmf = tf.constant([p**i * (1-p)**(n - i) * binom(n, i) for i in range(n + 1)])

d2l.plt.stem([i for i in range(n + 1)], pmf, use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```

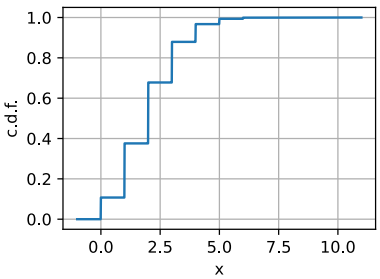


では、累積分布関数 (25.8.10) を描いてみよう。

```
x = torch.arange(-1, 11, 0.01)
cmf = torch.cumsum(pmf, dim=0)

def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]

d2l.plot(x, torch.tensor([F(y) for y in x.tolist()]), 'x', 'c.d.f.')
```



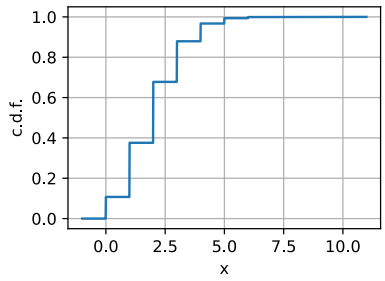

```
x = np.arange(-1, 11, 0.01)
cmf = np.cumsum(pmf)

def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]
```

(continues on next page)

(continued from previous page)

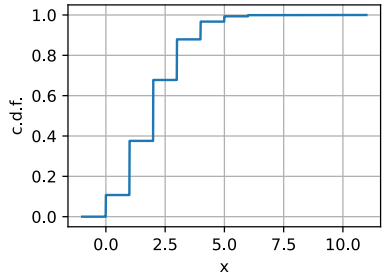
```
d2l.plot(x, np.array([F(y) for y in x.tolist()]), 'x', 'c.d.f.')
```




```
x = tf.range(-1, 11, 0.01)
cmf = tf.cumsum(pmf)

def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]

d2l.plot(x, [F(y) for y in x.numpy().tolist()], 'x', 'c.d.f.')
```



$X \sim \text{Binomial}(n, p)$ ならば、

- $\mu_X = np$、
- $\sigma_X^2 = np(1-p)$。

これは、n 個のベルヌーイ確率変数の和に対する期待値の線形性と、独立な確率変数の和の分散が分散の和になることから従う。これは次のようにサンプルできる。

```
m = torch.distributions.binomial.Binomial(n, p)
m.sample(sample_shape=(10, 10))
```

```
tensor([[2., 2., 2., 1., 3., 1., 2., 2., 1., 2.],
```

(continues on next page)

(continued from previous page)

```
        [3., 3., 2., 1., 2., 4., 1., 3., 1., 2.],
        [1., 2., 4., 1., 4., 4., 1., 0., 0., 1.],
        [1., 1., 0., 4., 2., 2., 2., 1., 2., 2.],
        [2., 3., 2., 1., 0., 2., 4., 3., 1., 2.],
        [1., 1., 5., 2., 3., 2., 3., 1., 3., 2.],
        [1., 6., 2., 1., 0., 3., 4., 4., 2., 4.],
        [2., 3., 2., 1., 2., 2., 3., 2., 2., 3.],
        [2., 1., 5., 0., 1., 3., 1., 2., 2., 5.],
        [0., 2., 5., 0., 0., 2., 3., 3., 0., 2.]])
```

```
np.random.binomial(n, p, size=(10, 10))
```

```
array([[5, 3, 2, 1, 3, 1, 1, 1, 3, 1],
       [2, 2, 2, 0, 1, 1, 1, 3, 0, 3],
       [1, 2, 3, 2, 1, 1, 2, 2, 2, 1],
       [1, 1, 1, 2, 2, 2, 0, 3, 4, 1],
       [2, 1, 2, 3, 1, 2, 1, 5, 2, 4],
       [4, 3, 2, 4, 0, 1, 2, 4, 3, 2],
       [1, 3, 1, 4, 1, 0, 2, 2, 3, 0],
       [2, 3, 1, 1, 3, 4, 3, 2, 1, 3],
       [5, 1, 4, 2, 2, 2, 0, 0, 2, 4],
       [3, 1, 3, 2, 0, 6, 2, 0, 2, 1]])
```

```
m = tfp.distributions.Binomial(n, p)
m.sample(sample_shape=(10, 10))
```

```
WARNING:tensorflow:From /home/ci/.local/lib/python3.10/site-packages/
↪tensorflow_probability/python/internal/batched_rejection_sampler.py:102:␣
↪calling while_loop_v2 (from tensorflow.python.ops.control_flow_ops) with␣
↪back_prop=False is deprecated and will be removed in a future version.
Instructions for updating:
back_prop=False is deprecated. Consider using tf.stop_gradient instead.
Instead of:
results = tf.while_loop(c, b, vars, back_prop=False)
Use:
results = tf.nest.map_structure(tf.stop_gradient, tf.while_loop(c, b, vars))
```

```
<tf.Tensor: shape=(10, 10), dtype=float32, numpy=
array([[4., 6., 7., 5., 6., 6., 5., 6., 6., 5.],
       [6., 7., 6., 6., 6., 4., 7., 3., 4., 7.],
       [5., 5., 6., 6., 6., 5., 6., 5., 5., 3.],
       [6., 4., 6., 7., 8., 4., 8., 5., 6., 7.],
       [5., 8., 6., 3., 6., 7., 4., 6., 7., 7.],
       [7., 8., 4., 5., 4., 6., 5., 5., 9., 5.],
       [7., 5., 4., 6., 9., 7., 8., 7., 5., 6.],
       [6., 6., 5., 6., 5., 4., 4., 6., 5., 8.],
       [7., 5., 5., 3., 6., 6., 6., 4., 8., 7.],
       [7., 3., 7., 5., 7., 7., 7., 4., 5., 6.]], dtype=float32)>
```

25.8.5 ポアソン分布

では、思考実験をしてみよう。私たちはバス停に立っていて、次の 1 分間に何台のバスが到着するかを知りたいとする。まず、$X^{(1)} \sim \text{Bernoulli}(p)$ を考える。これは、1 分間の窓の中でバスが到着する確率そのものである。都市中心部から離れたバス停では、これはかなり良い近似かもしれない。1 分間に 2 台以上のバスを見ることはまずないだろう。

しかし、混雑した地域にいるなら、2 台のバスが到着することもありえますし、むしろ起こりやすいかもしれない。これを、最初の 30 秒と次の 30 秒に分けて確率変数を分割することでモデル化できる。この場合、

$$X^{(2)} \sim X_1^{(2)} + X_2^{(2)}, \tag{25.8.11}$$

と書ける。ここで $X^{(2)}$ は合計であり、$X_i^{(2)} \sim \text{Bernoulli}(p/2)$ である。すると全体の分布は $X^{(2)} \sim \text{Binomial}(2, p/2)$ になる。

ここで止める必要はあるだろうか。さらにその 1 分を n 個に分割してみよう。上と同じ推論により、

$$X^{(n)} \sim \text{Binomial}(n, p/n). \tag{25.8.12}$$

が得られる。

これらの確率変数を考える。前節より、(25.8.12) の平均は $\mu_{X^{(n)}} = n(p/n) = p$、分散は $\sigma^2_{X^{(n)}} = n(p/n)(1-(p/n)) = p(1-p/n)$ である。$n \to \infty$ とすると、これらの値は $\mu_{X^{(\infty)}} = p$、分散 $\sigma^2_{X^{(\infty)}} = p$ に安定することがわかる。これは、この無限分割の極限で定義できる確率変数が 存在するかもしれないことを示している。

現実世界ではバスの到着回数を数えればよいだけなので、これはそれほど驚くことではないが、数学的モデルがきちんと定義されていることを見るのは良いことである。この議論は 稀な事象の法則として形式化できる。

この推論を丁寧にたどると、次のモデルに到達できる。$X \sim \text{Poisson}(\lambda)$ とは、$\{0, 1, 2, \ldots\}$ の値を確率

$$p_k = \frac{\lambda^k e^{-\lambda}}{k!}. \tag{25.8.13}$$

でとる確率変数のことである。

$\lambda > 0$ は 率（あるいは 形状パラメータ）と呼ばれ、1 単位時間あたりに期待される到着回数を表する。

この確率質量関数を和をとって累積分布関数を得ることができる。

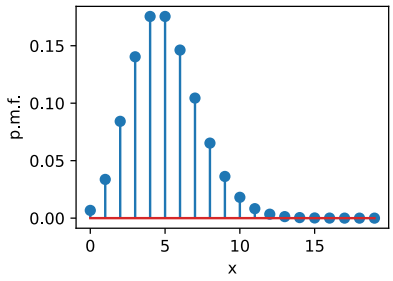
$$F(x) = \begin{cases} 0 & x < 0, \\ e^{-\lambda} \sum_{m=0}^{k} \frac{\lambda^m}{m!} & k \le x < k+1 \text{ with } 0 \le k. \end{cases} \tag{25.8.14}$$

まず確率質量関数 (25.8.13) を描きよう。

```
lam = 5.0

xs = [i for i in range(20)]
pmf = torch.tensor([torch.exp(torch.tensor(-lam)) * lam**k
                    / factorial(k) for k in xs])

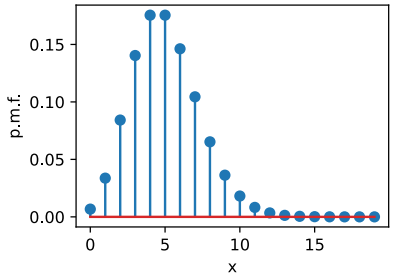
d2l.plt.stem(xs, pmf, use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```




```
lam = 5.0

xs = [i for i in range(20)]
pmf = np.array([np.exp(-lam) * lam**k / factorial(k) for k in xs])

d2l.plt.stem(xs, pmf, use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```




```
lam = 5.0

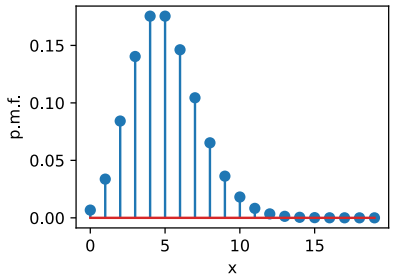
xs = [i for i in range(20)]
pmf = tf.constant([tf.exp(tf.constant(-lam)).numpy() * lam**k
                    / factorial(k) for k in xs])

d2l.plt.stem(xs, pmf, use_line_collection=True)
```

(continues on next page)

(continued from previous page)

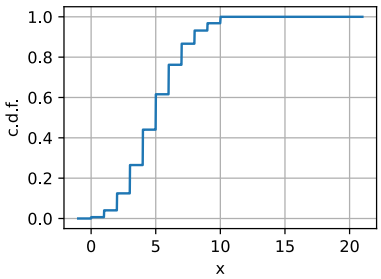
```
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```



では、累積分布関数 (25.8.14) を描いてみよう。

```
x = torch.arange(-1, 21, 0.01)
cmf = torch.cumsum(pmf, dim=0)
def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]

d2l.plot(x, torch.tensor([F(y) for y in x.tolist()]), 'x', 'c.d.f.')
```

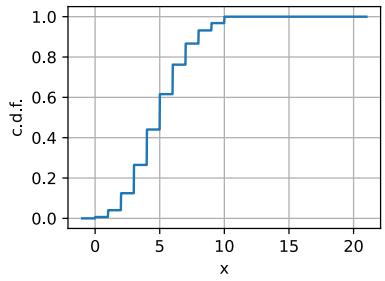


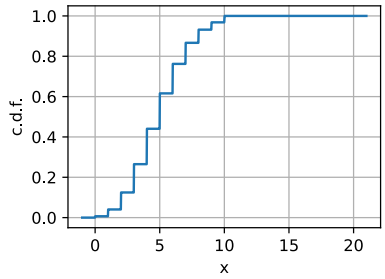

```
x = np.arange(-1, 21, 0.01)
cmf = np.cumsum(pmf)
def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]

d2l.plot(x, np.array([F(y) for y in x.tolist()]), 'x', 'c.d.f.')
```

```
x = tf.range(-1, 21, 0.01)
cmf = tf.cumsum(pmf)
def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]

d2l.plot(x, [F(y) for y in x.numpy().tolist()], 'x', 'c.d.f.')
```





上で見たように、平均と分散は特に簡潔である。$X \sim \text{Poisson}(\lambda)$ ならば、

- $\mu_X = \lambda$、
- $\sigma_X^2 = \lambda$。

これは次のようにサンプルできる。

```
m = torch.distributions.poisson.Poisson(lam)
m.sample((10, 10))
```

```
tensor([[ 1.,  7.,  6.,  3.,  1.,  4.,  3.,  5.,  6.,  5.],
        [ 3.,  5.,  4.,  5.,  8.,  8.,  5.,  2.,  6.,  8.],
        [ 6.,  7.,  7.,  6.,  8.,  5.,  6.,  6.,  4.,  4.],
        [ 4.,  6.,  5.,  3.,  3.,  4.,  8.,  8.,  5.,  5.],
        [ 3.,  7.,  7.,  6.,  6.,  3.,  8.,  8.,  7.,  3.],
        [ 7.,  4.,  4.,  8.,  5.,  3.,  4.,  5.,  5.,  3.],
        [ 4.,  7.,  3.,  2.,  5.,  9.,  4.,  6.,  4.,  8.],
        [ 5.,  5.,  4.,  4.,  4.,  4.,  3.,  7.,  2.,  6.],
        [ 4.,  8., 10.,  4.,  6.,  7.,  5., 10.,  5.,  5.],
        [ 2.,  5.,  4.,  5.,  7.,  4.,  2.,  4.,  5.,  8.]])
```

```
np.random.poisson(lam, size=(10, 10))
```

```
array([[ 5,  5,  6,  4,  2,  4,  8,  3,  4,  7],
       [ 0,  3, 11,  5,  3,  2,  7,  8,  3,  1],
       [ 6,  5,  4,  9,  7,  2,  1,  6,  6,  5],
```

(continues on next page)

(continued from previous page)

```
       [ 6,  5,  5,  3,  6,  4,  5,  3,  6,  4],
       [ 6,  6,  5,  5,  5,  5,  5,  9,  7,  7],
       [ 5,  4,  4,  6,  2,  5,  4,  7,  7,  2],
       [ 1,  4,  7,  2,  4,  8,  1,  4,  6,  4],
       [ 3,  4,  6,  8,  8,  4,  3,  4,  5,  3],
       [ 4,  6,  6, 10,  6,  4,  6,  3,  8,  2],
       [ 2,  9,  8,  3,  7,  5,  6,  4,  5,  3]])
```

```
m = tfp.distributions.Poisson(lam)
m.sample((10, 10))
```

```
<tf.Tensor: shape=(10, 10), dtype=float32, numpy=
array([[ 5.,  4.,  3.,  4.,  3.,  6.,  3., 10.,  5.,  4.],
       [ 9.,  7.,  3.,  3.,  5.,  3.,  5.,  5.,  3.,  1.],
       [ 7.,  6.,  5.,  6.,  8.,  3.,  2.,  4.,  4.,  3.],
       [ 7.,  2.,  4.,  4.,  5.,  9.,  2.,  5.,  6.,  6.],
       [ 6.,  8.,  3.,  3.,  4.,  5.,  6.,  5.,  3.,  6.],
       [ 3.,  3.,  6.,  5.,  3.,  0.,  5.,  8.,  5.,  7.],
       [ 2.,  4.,  1.,  9.,  8.,  8.,  7.,  5.,  5.,  6.],
       [ 5.,  4.,  6., 10.,  4., 10.,  8.,  6.,  3.,  6.],
       [ 4.,  3.,  5.,  4.,  5.,  6.,  4.,  7.,  7.,  2.],
       [ 5.,  4.,  5.,  5.,  5.,  5.,  6.,  5.,  4.,  3.]], dtype=float32)>
```

25.8.6 ガウス分布

では、別の、しかし関連した実験をしてみよう。ここでも、n 個の独立な Bernoulli(p) 測定 X_i を行うとする。これらの和の分布は $X^{(n)} \sim \text{Binomial}(n, p)$ である。n を増やし p を減らす極限をとる代わりに、p を固定して $n \to \infty$ としてみよう。この場合、$\mu_{X^{(n)}} = np \to \infty$ かつ $\sigma^2_{X^{(n)}} = np(1-p) \to \infty$ となるので、この極限がうまく定義されるとは考えにくいである。

しかし、希望はまだある。平均と分散がうまく振る舞うように、次を定義しよう。

$$Y^{(n)} = \frac{X^{(n)} - \mu_{X^{(n)}}}{\sigma_{X^{(n)}}}. \tag{25.8.15}$$

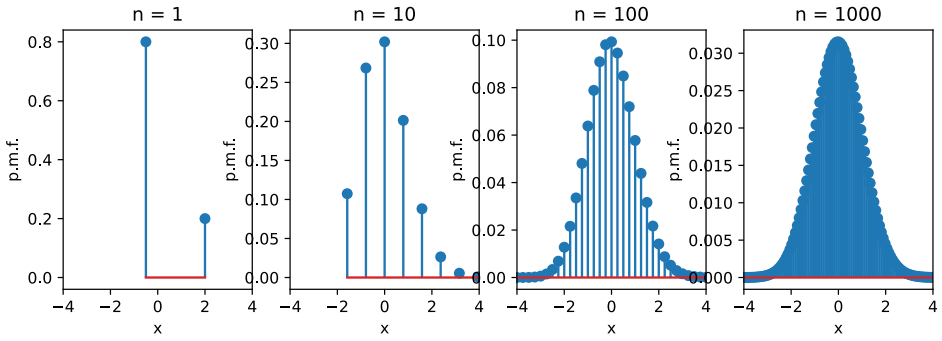
これは平均 0、分散 1 を持つことがわかるので、何らかの極限分布に収束すると考えるのはもっともらしいである。これらの分布がどのように見えるかを描いてみると、さらにそれがうまくいきそうだと確信できるだろう。

```
p = 0.2
ns = [1, 10, 100, 1000]
d2l.plt.figure(figsize=(10, 3))
for i in range(4):
    n = ns[i]
    pmf = torch.tensor([p**i * (1-p)**(n-i) * binom(n, i)
                        for i in range(n + 1)])
```

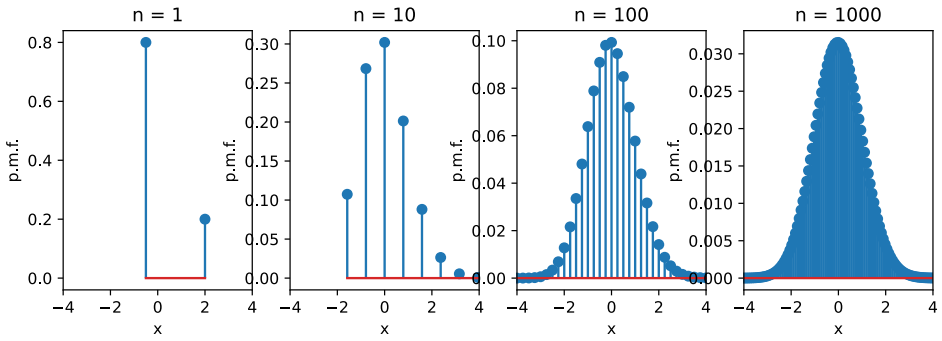
(continues on next page)

(continued from previous page)

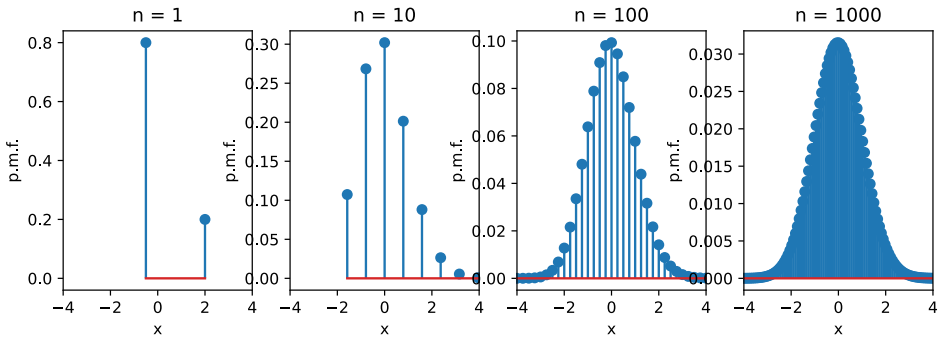
```
    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.stem([(i - n*p)/torch.sqrt(torch.tensor(n*p*(1 - p)))
                  for i in range(n + 1)], pmf,
                 use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')
    d2l.plt.title("n = {}".format(n))
d2l.plt.show()
```




```
p = 0.2
ns = [1, 10, 100, 1000]
d2l.plt.figure(figsize=(10, 3))
for i in range(4):
    n = ns[i]
    pmf = np.array([p**i * (1-p)**(n-i) * binom(n, i) for i in range(n + 1)])
    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.stem([(i - n*p)/np.sqrt(n*p*(1 - p)) for i in range(n + 1)], pmf,
                 use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')
    d2l.plt.title("n = {}".format(n))
d2l.plt.show()
```



```
p = 0.2
ns = [1, 10, 100, 1000]
d2l.plt.figure(figsize=(10, 3))
for i in range(4):
    n = ns[i]
    pmf = tf.constant([p**i * (1-p)**(n-i) * binom(n, i)
                       for i in range(n + 1)])
    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.stem([(i - n*p)/tf.sqrt(tf.constant(n*p*(1 - p)))
                  for i in range(n + 1)], pmf,
                 use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')
    d2l.plt.title("n = {}".format(n))
d2l.plt.show()
```



注意すべき点が 1 つある。ポアソンの場合と比べると、ここでは標準偏差で割っているため、取りうる結果をより狭い領域に押し込めている。これは、この極限がもはや離散的ではなく、連続的になることを示している。

何が起こるかの導出はこの文書の範囲を超えるが、中心極限定理によれば、$n \to \infty$ のとき、これはガウス分布（あるいは正規分布）を与える。より明示的には、任意の a, b に対して

$$\lim_{n\to\infty} P(Y^{(n)} \in [a, b]) = P(\mathcal{N}(0, 1) \in [a, b]), \tag{25.8.16}$$

が成り立ちる。ここで、確率変数が平均 μ、分散 σ^2 を持つ正規分布に従うとき、$X \sim \mathcal{N}(\mu, \sigma^2)$ と書き、その密度は

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{25.8.17}$$

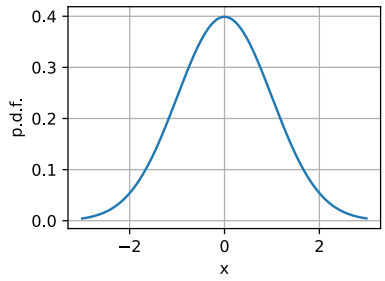
である。

まず確率密度関数 (25.8.17) を描きよう。

```
mu, sigma = 0, 1

x = torch.arange(-3, 3, 0.01)
p = 1 / torch.sqrt(2 * torch.pi * sigma**2) * torch.exp(
    -(x - mu)**2 / (2 * sigma**2))

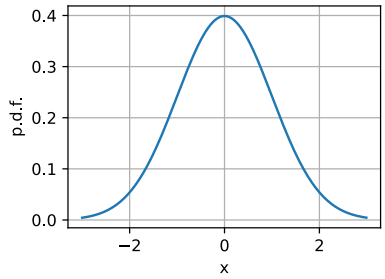
d2l.plot(x, p, 'x', 'p.d.f.')
```




```
mu, sigma = 0, 1

x = np.arange(-3, 3, 0.01)
p = 1 / np.sqrt(2 * np.pi * sigma**2) * np.exp(-(x - mu)**2 / (2 * sigma**2))

d2l.plot(x, p, 'x', 'p.d.f.')
```

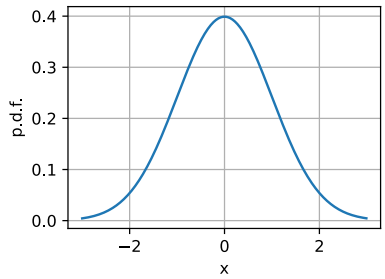



```
mu, sigma = 0, 1

x = tf.range(-3, 3, 0.01)
p = 1 / tf.sqrt(2 * tf.pi * sigma**2) * tf.exp(
    -(x - mu)**2 / (2 * sigma**2))

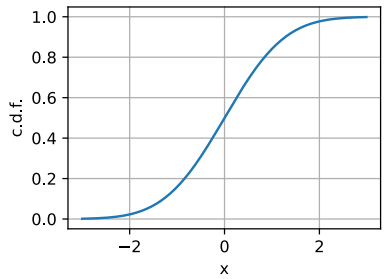
d2l.plot(x, p, 'x', 'p.d.f.')
```

では、累積分布関数を描いてみよう。ここでは詳しく述べないが、ガウス分布の c.d.f. は、より初等的な関数を用いた閉じた形の式を持たない。そこで、`erf` を使ってこの積分を数値的に計算する。



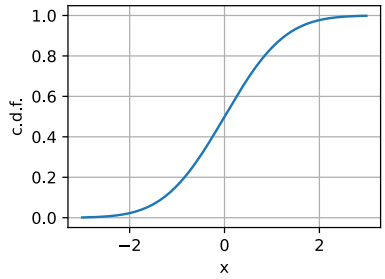

```
def phi(x):
    return (1.0 + erf((x - mu) / (sigma * torch.sqrt(d2l.tensor(2.)))) / 2.0

d2l.plot(x, torch.tensor([phi(y) for y in x.tolist()]), 'x', 'c.d.f.')
```




```
def phi(x):
    return (1.0 + erf((x - mu) / (sigma * np.sqrt(2)))) / 2.0

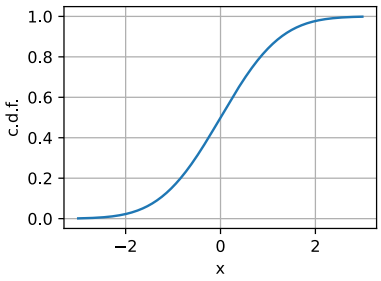
d2l.plot(x, np.array([phi(y) for y in x.tolist()]), 'x', 'c.d.f.')
```




```
def phi(x):
    return (1.0 + erf((x - mu) / (sigma * tf.sqrt(tf.constant(2.)))) / 2.0

d2l.plot(x, [phi(y) for y in x.numpy().tolist()], 'x', 'c.d.f.')
```

注意深い読者なら、これらの項のいくつかに見覚えがあるだろう。実際、この



積分は Section 25.5 で扱いた。まさにその計算によって、この $p_X(x)$ の全体の面積が 1 であり、したがって有効な密度であることがわかる。

コイン投げを用いたのは計算を簡単にするためでしたが、その選択自体に本質的な意味はない。実際、独立同分布な確率変数 X_i の任意の集まりを取り、

$$X^{(N)} = \sum_{i=1}^{N} X_i. \tag{25.8.18}$$

としたとき、

$$\frac{X^{(N)} - \mu_{X^{(N)}}}{\sigma_{X^{(N)}}} \tag{25.8.19}$$

はおおよそガウス分布に従う。うまく成り立つためには追加の条件が必要で、最も一般的には $E[X^4] < \infty$ であるが、考え方は明確である。

中心極限定理は、ガウス分布が確率、統計、機械学習において基本的である理由である。何か測定したものが多くの小さな独立な寄与の和だと言えるなら、その測定対象はガウス分布に近いと仮定できる。

ガウス分布にはさらに多くの興味深い性質があるが、ここではもう 1 つだけ述べよう。ガウス分布は 最大エントロピー分布として知られている。エントロピーについては Section 25.11 でより深く扱うが、ここで知っておくべきことは、それがランダムさの尺度であるということだけである。厳密な数学的意味では、ガウス分布は平均と分散が固定された確率変数の中で 最もランダムな選択だと考えられる。したがって、確率変数の平均と分散がわかっているなら、ガウス分布はある意味で最も保守的な分布の選択である。

節を閉じる前に、$X \sim \mathcal{N}(\mu, \sigma^2)$ ならば、

- $\mu_X = \mu$、
- $\sigma_X^2 = \sigma^2$。

であることを思い出そう。

ガウス分布（あるいは標準正規分布）からのサンプルは以下のように得られる。

```
torch.normal(mu, sigma, size=(10, 10))
```

```
tensor([[ 5.6402e-01, -2.9638e+00, -4.0225e-01, -1.0502e+00, -8.5967e-01,
         -1.8663e+00,  1.2591e+00, -4.8977e-01, -1.8643e-01, -8.4658e-01],
        [ 9.8634e-01,  2.5091e-01,  5.5886e-01, -4.9260e-01, -1.7324e-01,
         -4.1209e-01, -5.2245e-01,  5.0280e-01, -6.3664e-01, -9.0722e-01],
        [-1.1246e+00, -1.2719e+00, -1.0566e+00,  1.5562e+00, -3.7938e-01,
          8.7909e-02,  1.5184e+00, -6.9584e-01, -1.4542e+00, -2.9577e+00],
        [-1.3921e-02,  1.1291e+00,  2.1045e-01,  1.5291e+00,  2.8422e-02,
          1.3347e+00,  2.0202e+00,  2.0668e-01,  8.9387e-01,  2.1965e-03],
        [ 1.3032e+00,  8.9880e-01, -3.0384e-01,  7.6841e-01,  4.2073e-02,
         -3.1455e-01, -1.5442e+00, -1.0123e-01, -2.5143e-01, -1.8960e+00],
        [-2.0620e+00,  7.7326e-01, -9.0571e-02,  6.0874e-01,  3.1394e-01,
         -3.9894e-01, -3.0525e-01, -9.5778e-01,  1.6019e-01, -5.7222e-02],
        [-5.1813e-01, -1.4124e-01,  4.8078e-01,  1.9416e+00,  8.5896e-01,
          5.8826e-01,  6.5373e-01,  5.8058e-01, -1.4395e+00,  4.8357e-01],
        [-1.5158e+00, -2.5533e-01, -6.5353e-01,  1.0617e+00,  4.6570e-01,
          1.8963e-01,  1.6243e+00,  5.5373e-01,  1.8233e-01,  1.3987e+00],
        [ 1.7548e+00, -5.5406e-02,  7.8020e-01, -1.9451e+00, -1.1238e+00,
          1.3765e+00,  4.4821e-01, -8.6358e-01, -7.7124e-01, -1.8096e-01],
        [ 8.7309e-01,  6.8673e-01,  6.5082e-01,  1.7901e+00, -6.1880e-01,
         -3.6823e-01,  3.9230e-01, -7.9143e-01,  8.1201e-01, -2.5098e-01]])
```

```
np.random.normal(mu, sigma, size=(10, 10))
```

```
array([[-0.70057691, -1.08498021,  0.61045712,  0.38959336, -0.39842221,
         0.44038231,  0.71311424,  0.87899763, -1.04249347, -1.99314222],
       [-1.19042449,  1.42323131, -2.03115771, -0.55207844, -0.34944086,
        -0.88541313,  0.68772012, -1.08336504,  0.0043346 , -0.84555549],
       [-1.01220272, -0.18602191, -0.96093954, -0.06832284,  0.67980878,
        -0.84498207,  0.12785912,  0.42840493, -1.02172581,  0.94296419],
       [-1.47794148, -1.0995401 ,  0.32091134, -1.38076821,  0.53228056,
         1.37388727,  0.85008316, -0.8838563 ,  0.25436875,  0.8664244 ],
       [-0.86416781,  0.68150441,  0.91070442, -0.31040679, -1.24914887,
        -1.09228335,  0.23755172,  0.46968028, -1.38901965, -1.3061524 ],
       [ 0.92143486,  2.25051919,  0.93860495, -0.04962321, -0.94616495,
        -0.30157155,  0.25928661,  0.52036172, -0.33842228, -1.16051394],
       [ 0.39067333, -0.29209444,  2.04488421,  1.26964076,  0.02560868,
         0.09079073, -0.89648959, -0.805082  ,  0.0373519 ,  1.77227377],
       [-1.30955062,  1.55305581, -0.87499763, -0.37879967,  1.41977948,
        -1.56414618, -0.19597413,  0.17618799,  1.39520775,  0.8540339 ],
       [ 2.00466515,  0.78036987,  1.50414668,  0.8129593 , -0.33577407,
        -0.74829381, -0.51340285, -0.16907909,  0.26541118, -0.52972822],
       [-0.12352982,  1.31952519, -0.64136017, -0.11811265,  1.37662606,
        -0.09667546,  0.41606841, -0.06347404,  0.78929749, -0.40645727]])
```

```
tf.random.normal((10, 10), mu, sigma)
```

```
<tf.Tensor: shape=(10, 10), dtype=float32, numpy=
array([[ 1.7760942 , -0.4816362 , -0.35947865, -1.9446675 , -1.0896894 ,
```

(continues on next page)

(continued from previous page)

```
        1.5467461 , -1.2217262 ,  1.1850185 ,  0.8834578 ,  0.06816649],
       [-0.6936654 ,  1.2579052 ,  0.588886  , -0.82313985, -0.4076043 ,
        0.8245478 , -0.4964524 ,  0.39823017,  0.6062789 ,  0.5577762 ],
       [ 0.25759804, -1.9598231 , -0.26861465, -0.34204927,  0.91185987,
        -1.2176397 ,  0.72053534,  0.67268974, -1.6410838 , -0.4068907 ],
       [-0.46769965,  0.75001246,  0.14487791, -0.98179877, -1.1259598 ,
        1.1308956 , -0.5497258 ,  0.3834835 ,  0.9722934 ,  0.6679007 ],
       [-0.92601836, -1.2266558 ,  0.274593  ,  0.43597487,  0.52778405,
        -0.32230726,  0.25692734, -1.5467511 ,  0.26726183, -1.2071371 ],
       [ 1.1377119 ,  1.420974  ,  0.3854779 , -0.30418903, -2.0691662 ,
        1.4372991 ,  0.41003123, -0.18349372,  0.4678633 ,  1.1770911 ],
       [ 0.7949645 , -0.24854268, -0.2203115 , -0.10029124, -0.17249188,
        0.59759915,  2.479362  ,  0.3725324 ,  0.0325313 ,  0.25079554],
       [ 0.44387078,  0.60749555, -0.29627955, -0.93514496, -0.7731776 ,
        -1.6067672 ,  2.6808097 , -2.1464765 ,  0.03318942,  0.9157554 ],
       [-0.59047383,  0.6210598 , -0.28843474, -0.20017219,  1.0729624 ,
        1.8634268 , -0.36915967, -0.53046304, -1.0391091 , -0.6351748 ],
       [-0.73569167, -0.7207913 , -1.1421863 ,  0.47300977, -0.03475155,
        -1.6653922 ,  1.6053184 , -1.5436187 , -0.03438377,  0.4566425 ]],
      dtype=float32)>
```

25.8.7 指数型分布族

上で挙げた分布に共通する性質の 1 つは、すべてが 指数型分布族に属することである。指数型分布族は、密度を次の形で表せる分布の集合である。

$$p(\mathbf{x} \mid \boldsymbol{\eta}) = h(\mathbf{x}) \cdot \exp\left(\boldsymbol{\eta}^\top \cdot T(\mathbf{x}) - A(\boldsymbol{\eta})\right) \tag{25.8.20}$$

この定義は少し繊細なので、詳しく見ていきよう。

まず、$h(\mathbf{x})$ は 基底測度あるいは ベース測度と呼ばれる。これは、指数重みで修正する元の測度とみなせる。

次に、$\boldsymbol{\eta} = (\eta_1, \eta_2, ..., \eta_l) \in \mathbb{R}^l$ というベクトルがあり、これを 自然パラメータまたは 標準パラメータと呼ぶ。これらは、基底測度をどのように修正するかを定める。自然パラメータは、$\mathbf{x} = (x_1, x_2, ..., x_n) \in \mathbb{R}^n$ のある関数 $T(\cdot)$ に対してこれらのパラメータとの内積を取り、それを指数化することで新しい測度に入る。ベクトル $T(\mathbf{x}) = (T_1(\mathbf{x}), T_2(\mathbf{x}), ..., T_l(\mathbf{x}))$ は、$\boldsymbol{\eta}$ に対する 十分統計量と呼ばれる。この名前は、$T(\mathbf{x})$ に表される情報だけで確率密度を計算するのに十分であり、サンプル $\mathbf{x}$ の他の情報は不要だからである。

第三に、$A(\boldsymbol{\eta})$ がある。これは 累積関数と呼ばれ、上の分布 (25.8.20) が 1 に積分されること、すなわち

$$A(\boldsymbol{\eta}) = \log\left[\int h(\mathbf{x}) \cdot \exp\left(\boldsymbol{\eta}^\top \cdot T(\mathbf{x})\right) d\mathbf{x}\right]. \tag{25.8.21}$$

を保証する。

具体的に、ガウス分布を考えてみよう。$\mathbf{x}$ が 1 変数であるとすると、その密度

は

$$
\begin{aligned}
p(x \mid \mu, \sigma) &= \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left\{\frac{-(x-\mu)^2}{2\sigma^2}\right\} \\
&= \frac{1}{\sqrt{2\pi}} \cdot \exp\left\{\frac{\mu}{\sigma^2}x - \frac{1}{2\sigma^2}x^2 - \left(\frac{1}{2\sigma^2}\mu^2 + \log(\sigma)\right)\right\}.
\end{aligned}
\tag{25.8.22}
$$

これは指数型分布族の定義に次のように対応する。

- 基底測度: $h(x) = \frac{1}{\sqrt{2\pi}}$、
- 自然パラメータ: $\boldsymbol{\eta} = \begin{bmatrix}\eta_1 \\ \eta_2\end{bmatrix} = \begin{bmatrix}\frac{\mu}{\sigma^2} \\ \frac{1}{2\sigma^2}\end{bmatrix}$、
- 十分統計量: $T(x) = \begin{bmatrix}x \\ -x^2\end{bmatrix}$、および
- 累積関数: $A(\boldsymbol{\eta}) = \frac{1}{2\sigma^2}\mu^2 + \log(\sigma) = \frac{\eta_1^2}{4\eta_2} - \frac{1}{2}\log(2\eta_2)$。

上の各項の正確な選び方は、ある程度任意であることに注意する価値がある。重要なのは、分布がこの形で表せることであって、その厳密な形そのものではない。

Section 7.1.2 でほのめかしたように、広く使われる手法の 1 つは、最終出力 $\mathbf{y}$ が指数型分布族に従うと仮定することである。指数型分布族は、機械学習で頻繁に現れる、一般的で強力な分布族である。

25.8.8 まとめ

- ベルヌーイ確率変数は、はい/いいえの結果を持つ事象をモデル化できる。
- 離散一様分布は、有限個の候補からの選択をモデル化する。
- 連続一様分布は、区間からの選択をモデル化する。
- 二項分布は、ベルヌーイ確率変数の系列をモデル化し、成功回数を数える。
- ポアソン確率変数は、稀な事象の到着をモデル化する。
- ガウス確率変数は、多数の独立な確率変数を足し合わせた結果をモデル化する。
- 上記の分布はすべて指数型分布族に属する。

25.8.9 演習

1. 独立な二項確率変数 $X, Y \sim \text{Binomial}(16, 1/2)$ の差 $X - Y$ である確率変数の標準偏差はいくつか。
2. ポアソン確率変数 $X \sim \text{Poisson}(\lambda)$ を取り、$\lambda \to \infty$ のときの $(X - \lambda)/\sqrt{\lambda}$ を考えると、これが近似的にガウス分布になることを示せる。なぜこれは自然なのか。

3. n 個の要素を持つ 2 つの離散一様確率変数の和の確率質量関数は何か。

25.9 ナイーブベイズ

これまでの節を通して、確率論と確率変数について学んできた。この理論を実際に役立てるために、ナイーブベイズ分類器を導入しよう。これは確率の基本原理だけを用いて、数字の分類を可能にする。

学習とは、仮定を置くことにほかならない。これまで見たことのない新しいデータ例を分類したいなら、どのデータ例が互いに似ているかについて何らかの仮定を置かなければならない。ナイーブベイズ分類器は、広く使われている非常に明快なアルゴリズムで、計算を簡単にするために、すべての特徴が互いに独立であると仮定する。この節では、このモデルを画像中の文字認識に適用する。

```
%matplotlib inline
from d2l import torch as d2l
import math
import torch
import torchvision
d2l.use_svg_display()
```

```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import gluon, np, npx
npx.set_np()
d2l.use_svg_display()
```

```
image.shape, image.dtype
```

```
%matplotlib inline
from d2l import tensorflow as d2l
import math
import tensorflow as tf
d2l.use_svg_display()
```

25.9.1 光学文字認識

MNIST (LeCun *et al.*, 1998) は、広く使われているデータセットの一つである。これは学習用に 60,000 枚、検証用に 10,000 枚の画像を含む。各画像には 0 から 9 までの手書き数字が 1 つ含まれている。課題は、それぞれの画像を対応する数字に分類することである。

Gluon は data.vision モジュールに MNIST クラスを提供しており、データセットをインターネットから自動的に取得できる。その後は、Gluon はすでにダウンロード済みのローカルコピーを使用する。学習セットかテストセットかは、パラメータ train の値をそれぞれ True または False に設定することで指定する。各画像は幅と高さがともに 28 のグレースケール画像で、形状は (28,28,1) である。最後のチャネル次元を取り除くために、カスタマイズした変換を使う。さらに、このデータセットでは各画素が符号なし 8 ビット整数で表されている。問題を簡単にするために、これらを 2 値特徴に量子化する。

```
data_transform = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    lambda x: torch.floor(x * 255 / 128).squeeze(dim=0)
])

mnist_train = torchvision.datasets.MNIST(
    root='./temp', train=True, transform=data_transform, download=True)
mnist_test = torchvision.datasets.MNIST(
    root='./temp', train=False, transform=data_transform, download=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./
↪temp/MNIST/raw/train-images-idx3-ubyte.gz
100%|          | 9912422/9912422 [00:00<00:00, 119248043.86it/s]
Extracting ./temp/MNIST/raw/train-images-idx3-ubyte.gz to ./temp/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./
↪temp/MNIST/raw/train-labels-idx1-ubyte.gz
100%|          | 28881/28881 [00:00<00:00, 6166549.27it/s]
Extracting ./temp/MNIST/raw/train-labels-idx1-ubyte.gz to ./temp/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./
↪temp/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|          | 1648877/1648877 [00:00<00:00, 31792085.86it/s]
Extracting ./temp/MNIST/raw/t10k-images-idx3-ubyte.gz to ./temp/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./
↪temp/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|          | 4542/4542 [00:00<00:00, 6371414.30it/s]
Extracting ./temp/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./temp/MNIST/raw
```

```
def transform(data, label):
    return np.floor(data.astype('float32') / 128).squeeze(axis=-1), label

mnist_train = gluon.data.vision.MNIST(train=True, transform=transform)
mnist_test = gluon.data.vision.MNIST(train=False, transform=transform)
```

```
Downloading /opt/mxnet/datasets/mnist/train-images-idx3-ubyte.gz from https://
↪apache-mxnet.s3-accelerate.dualstack.amazonaws.com/gluon/dataset/mnist/train-
```

(continues on next page)

(continued from previous page)

```
↪images-idx3-ubyte.gz...
Downloading /opt/mxnet/datasets/mnist/train-labels-idx1-ubyte.gz from https://
↪apache-mxnet.s3-accelerate.dualstack.amazonaws.com/gluon/dataset/mnist/train-
↪labels-idx1-ubyte.gz...
[06:59:19] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
Downloading /opt/mxnet/datasets/mnist/t10k-images-idx3-ubyte.gz from https://
↪apache-mxnet.s3-accelerate.dualstack.amazonaws.com/gluon/dataset/mnist/t10k-
↪images-idx3-ubyte.gz...
Downloading /opt/mxnet/datasets/mnist/t10k-labels-idx1-ubyte.gz from https://
↪apache-mxnet.s3-accelerate.dualstack.amazonaws.com/gluon/dataset/mnist/t10k-
↪labels-idx1-ubyte.gz...
```

```
d2l.show_images(images, 2, 9);
```

```
((train_images, train_labels), (
    test_images, test_labels)) = tf.keras.datasets.mnist.load_data()

# Original pixel values of MNIST range from 0-255 (as the digits are stored as
# uint8). For this section, pixel values that are greater than 128 (in the
# original image) are converted to 1 and values that are less than 128 are
# converted to 0. See section 18.9.2 and 18.9.3 for why
train_images = tf.floor(tf.constant(train_images / 128, dtype = tf.float32))
test_images = tf.floor(tf.constant(test_images / 128, dtype = tf.float32))

train_labels = tf.constant(train_labels, dtype = tf.int32)
test_labels = tf.constant(test_labels, dtype = tf.int32)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
↪datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

画像と対応するラベルを含む、特定の例にアクセスできる。

```
image, label = mnist_train[2]
image.shape, label
```

```
(torch.Size([28, 28]), 4)
```

```
image, label = mnist_train[2]
image.shape, label
```

```
((28, 28), array(4, dtype=int32))
```

```
image, label = train_images[2], train_labels[2]
image.shape, label.numpy()
```

```
(TensorShape([28, 28]), 4)
```

ここで変数 image に格納されている例は、縦横ともに 28 ピクセルの画像に対応している。

```
image.shape, image.dtype
```

```
(torch.Size([28, 28]), torch.float32)
```

コードでは、各画像のラベルをスカラーとして保存している。その型は 32 ビット整数である。

```
label, type(label)
```

```
(4, int)
```

```
label, type(label), label.dtype
```

```
(array(4, dtype=int32), mxnet.numpy.ndarray, dtype('int32'))
```

```
label.numpy(), label.dtype
```

```
(4, tf.int32)
```

複数の例に同時にアクセスすることもできる。

```
images = torch.stack([mnist_train[i][0] for i in range(10, 38)], dim=0)
labels = torch.tensor([mnist_train[i][1] for i in range(10, 38)])
images.shape, labels.shape
```

```
(torch.Size([28, 28, 28]), torch.Size([28]))
```

```
images, labels = mnist_train[10:38]
images.shape, labels.shape
```

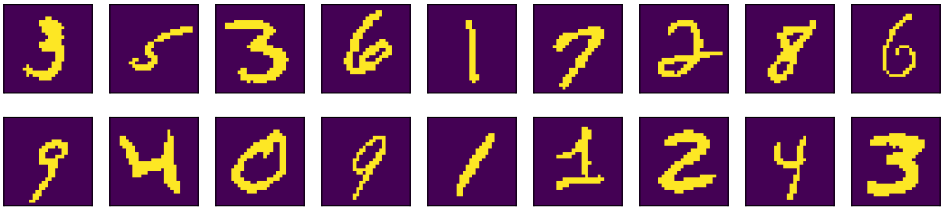
```
((28, 28, 28), (28,))
```

```
images = tf.stack([train_images[i] for i in range(10, 38)], axis=0)
labels = tf.constant([train_labels[i].numpy() for i in range(10, 38)])
images.shape, labels.shape
```

```
(TensorShape([28, 28, 28]), TensorShape([28]))
```

これらの例を可視化してみよう。

```
d2l.show_images(images, 2, 9);
```



25.9.2 分類のための確率モデル

分類タスクでは、例をあるカテゴリに写像する。ここでの例はグレースケールの 28×28 画像で、カテゴリは数字である。（より詳しい説明は Section 7.1 を参照する。）分類タスクを表現する自然な方法の一つは、確率的な問いとして考えることである。すなわち、特徴（つまり画像の画素）が与えられたとき、最もありそうなラベルは何か、という問いである。例の特徴を $\mathbf{x} \in \mathbb{R}^d$、ラベルを $y \in \mathbb{R}$ と表する。ここで特徴は画像の画素であり、2 次元画像をベクトルに変形すれば $d = 28^2 = 784$ となる。ラベルは数字である。特徴が与えられたときのラベルの確率は $p(y \mid \mathbf{x})$ である。もしこれらの確率、つまりこの例では $y = 0, \ldots, 9$ に対する $p(y \mid \mathbf{x})$ を計算できるなら、分類器は次の式で与えられる予測 $\hat{y}$ を出力する。

$$\hat{y} = \operatorname{argmax} p(y \mid \mathbf{x}). \tag{25.9.1}$$

残念ながら、これを行うには $\mathbf{x} = x_1, ..., x_d$ のすべての値に対して $p(y \mid \mathbf{x})$ を推定しなければならない。各特徴が 2 つの値のどちらかを取るとしよう。たとえば、特徴 $x_1 = 1$ はある文書に単語 apple が現れることを意味し、$x_1 = 0$ は現れないことを意味するとする。もしそのような 2 値特徴が 30 個あれば、入力ベクトル $\mathbf{x}$ の 2^{30}（10 億以上！）通りの可能な値のどれに対しても分類できるように準備しておく必要があることになる。

さらに、どこに学習があるのだろうか。対応するラベルを予測するためにあり得るすべての例を見なければならないなら、私たちは本当にパターンを学習しているのではなく、単にデータセットを記憶しているだけである。

25.9.3 ナイーブベイズ分類器

幸いなことに、条件付き独立についていくつかの仮定を置くことで、帰納バイアスを導入し、比較的少数の学習例から一般化できるモデルを構築できる。ま

ずベイズの定理を使って、分類器を次のように表しよう。

$$\hat{y} = \operatorname{argmax}_y p(y \mid \mathbf{x}) = \operatorname{argmax}_y \frac{p(\mathbf{x} \mid y)p(y)}{p(\mathbf{x})}. \tag{25.9.2}$$

分母は正規化項 $p(\mathbf{x})$ であり、ラベル y の値には依存しないことに注意する。その結果、異なる y の値に対して分子を比較することだけを考えれば十分である。たとえ分母の計算が困難であっても、分子を評価できる限り、それを無視して構わない。幸いなことに、正規化定数を復元したくなったとしても可能である。$\sum_y p(y \mid \mathbf{x}) = 1$ なので、正規化項は常に復元できる。

では、$p(\mathbf{x} \mid y)$ に注目しよう。確率の連鎖律を用いると、$p(\mathbf{x} \mid y)$ を次のように表せる。

$$p(x_1 \mid y) \cdot p(x_2 \mid x_1, y) \cdot \ldots \cdot p(x_d \mid x_1, \ldots, x_{d-1}, y). \tag{25.9.3}$$

これだけでは、まだ先に進めない。依然としておよそ 2^d 個のパラメータを推定しなければならない。しかし、ラベルが与えられたとき、特徴同士は条件付き独立であると仮定すれば、この項は $\prod_i p(x_i \mid y)$ に簡単化され、予測器は

$$\hat{y} = \operatorname{argmax}_y \prod_{i=1}^{d} p(x_i \mid y)p(y). \tag{25.9.4}$$

となる。

すべての i と y に対して $p(x_i = 1 \mid y)$ を推定し、その値を $P_{xy}[i, y]$ に保存できるとする。ここで P_{xy} は $d \times n$ 行列で、n はクラス数、$y \in \{1, \ldots, n\}$ である。すると、これを使って $p(x_i = 0 \mid y)$ も推定できる。すなわち、

$$p(x_i = t_i \mid y) = \begin{cases} P_{xy}[i, y] & \text{for } t_i = 1; \\ 1 - P_{xy}[i, y] & \text{for } t_i = 0. \end{cases} \tag{25.9.5}$$

さらに、各 y に対して $p(y)$ を推定して $P_y[y]$ に保存する。ここで P_y は長さ n のベクトルである。すると、新しい例 $\mathbf{t} = (t_1, t_2, \ldots, t_d)$ に対して、次を計算できる。

$$\begin{aligned} \hat{y} &= \operatorname{argmax}_y p(y) \prod_{i=1}^{d} p(x_t = t_i \mid y) \\ &= \operatorname{argmax}_y P_y[y] \prod_{i=1}^{d} P_{xy}[i, y]^{t_i} \left(1 - P_{xy}[i, y]\right)^{1 - t_i} \end{aligned} \tag{25.9.6}$$

任意の y に対してである。したがって、条件付き独立という仮定により、モデルの複雑さは特徴数に対する指数的依存 $O(2^d n)$ から線形依存 $O(dn)$ へと減少した。

25.9.4 学習

問題は、P_{xy} と P_y が分からないことである。したがって、まずいくつかの学習データからそれらの値を推定する必要がある。これがモデルの学習である。P_y の推定はそれほど難しくない。扱うクラスは 10 個しかないので、各

数字の出現回数 n_y を数え、全データ数 n で割ればよいのである。たとえば、数字 8 が $n_8 = 5,800$ 回現れ、画像の総数が $n = 60,000$ 枚なら、確率推定は $p(y = 8) = 0.0967$ である。

```
X = torch.stack([mnist_train[i][0] for i in range(len(mnist_train))], dim=0)
Y = torch.tensor([mnist_train[i][1] for i in range(len(mnist_train))])

n_y = torch.zeros(10)
for y in range(10):
    n_y[y] = (Y == y).sum()
P_y = n_y / n_y.sum()
P_y
```

```
tensor([0.0987, 0.1124, 0.0993, 0.1022, 0.0974, 0.0904, 0.0986, 0.1044, 0.0975,
        0.0992])
```

```
X, Y = mnist_train[:]  # All training examples

n_y = np.zeros((10))
for y in range(10):
    n_y[y] = (Y == y).sum()
P_y = n_y / n_y.sum()
P_y
```

```
array([0.09871667, 0.11236667, 0.0993    , 0.10218333, 0.09736667,
       0.09035   , 0.09863333, 0.10441667, 0.09751666, 0.09915   ])
```

```
X = train_images
Y = train_labels

n_y = tf.Variable(tf.zeros(10))
for y in range(10):
    n_y[y].assign(tf.reduce_sum(tf.cast(Y == y, tf.float32)))
P_y = n_y / tf.reduce_sum(n_y)
P_y
```

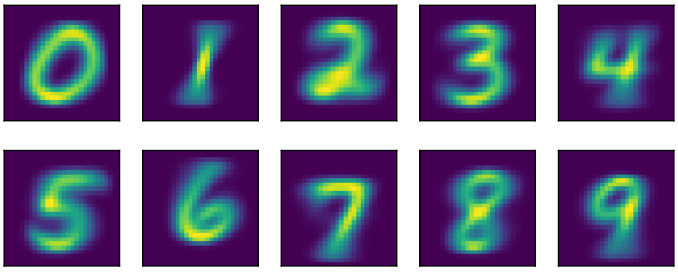
```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([0.09871667, 0.11236667, 0.0993    , 0.10218333, 0.09736667,
       0.09035   , 0.09863333, 0.10441667, 0.09751666, 0.09915   ],
      dtype=float32)>
```

次に、少し難しい P_{xy} に進む。白黒画像を選んだので、$p(x_i \mid y)$ は、クラス y に対して画素 i がオンになる確率を表する。先ほどと同様に、事象が起こる回数 n_{iy} を数え、それを y の総出現回数、すなわち n_y で割ればよいのである。しかし、少し厄介な点がある。ある画素が決して黒にならないことがあり得るのです（たとえば、きちんと切り抜かれた画像では、角の画素は常に白かもしれない）。統計学者がこの問題に対処する便利な方法は、すべての出現回数に擬似カウントを加えることである。したがって、n_{iy} の代わりに $n_{iy} + 1$ を使い、n_y の

代わりに $n_y + 2$ を使う（画素 i が取りうる値は 2 つ、つまり黒か白のどちらかだからです）。これは ラプラス平滑化とも呼ばれる。これは場当たり的に見えるかもしれないが、ベータ二項モデルによるベイズ的観点から動機づけることができる。

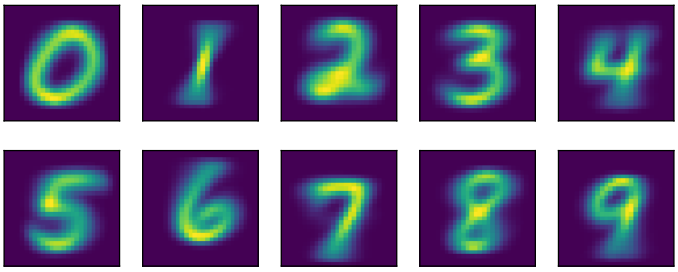
```
n_x = torch.zeros((10, 28, 28))
for y in range(10):
    n_x[y] = torch.tensor(X.numpy()[Y.numpy() == y].sum(axis=0))
P_xy = (n_x + 1) / (n_y + 2).reshape(10, 1, 1)

d2l.show_images(P_xy, 2, 5);
```



```
n_x = np.zeros((10, 28, 28))
for y in range(10):
    n_x[y] = np.array(X.asnumpy()[Y.asnumpy() == y].sum(axis=0))
P_xy = (n_x + 1) / (n_y + 2).reshape(10, 1, 1)

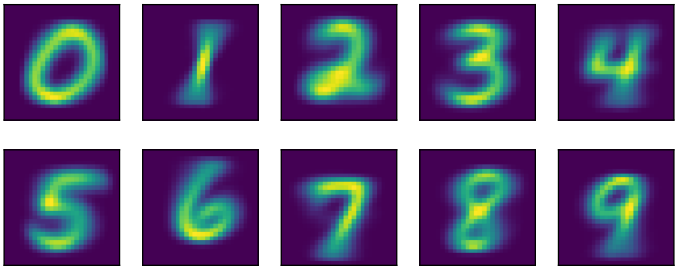
d2l.show_images(P_xy, 2, 5);
```



```
n_x = tf.Variable(tf.zeros((10, 28, 28)))
for y in range(10):
    n_x[y].assign(tf.cast(tf.reduce_sum(
        X.numpy()[Y.numpy() == y], axis=0), tf.float32))
P_xy = (n_x + 1) / tf.reshape((n_y + 2), (10, 1, 1))

d2l.show_images(P_xy, 2, 5);
```

これらの $10 \times 28 \times 28$ 個の確率（各クラスの各画素ごと）を可視化すると、何となく数字らしいものが見えてきる。



これで (25.9.6) を使って新しい画像を予測できる。$\mathbf{x}$ が与えられたとき、次の関数は各 y に対する $p(\mathbf{x} \mid y)p(y)$ を計算する。

```
def bayes_pred(x):
    x = x.unsqueeze(0)  # (28, 28) -> (1, 28, 28)
    p_xy = P_xy * x + (1 - P_xy)*(1 - x)
    p_xy = p_xy.reshape(10, -1).prod(dim=1)  # p(x|y)
    return p_xy * P_y

image, label = mnist_test[0]
bayes_pred(image)
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
def bayes_pred(x):
    x = np.expand_dims(x, axis=0)  # (28, 28) -> (1, 28, 28)
    p_xy = P_xy * x + (1 - P_xy)*(1 - x)
    p_xy = p_xy.reshape(10, -1).prod(axis=1)  # p(x|y)
    return np.array(p_xy) * P_y

image, label = mnist_test[0]
bayes_pred(image)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
def bayes_pred(x):
    x = tf.expand_dims(x, axis=0)  # (28, 28) -> (1, 28, 28)
    p_xy = P_xy * x + (1 - P_xy)*(1 - x)
    p_xy = tf.math.reduce_prod(tf.reshape(p_xy, (10, -1)), axis=1)  # p(x|y)
    return p_xy * P_y

image, label = train_images[0], train_labels[0]
bayes_pred(image)
```

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=array([0., 0., 0., 0., 0., 0., 0.
↪, 0., 0., 0.], dtype=float32)>
```

これはひどく失敗した！その理由を調べるために、画素ごとの確率を見てみよ

う。これらは通常 0.001 から 1 の間の数である。私たちはそれらを 784 個掛け合わせている。ここで重要なのは、これらの数をコンピュータ上で計算しているため、指数部の範囲が有限であるということである。起こるのは 数値アンダーフローで、小さな数をすべて掛け合わせるとさらに小さくなり、最終的には 0 に丸め込まれてしまいる。これは Section 25.7 で理論的な問題として議論したが、ここではその現象が実際に明確に見える。

その節で述べたように、$\log ab = \log a + \log b$ という性質を使って、対数の和を取るようにすれば解決できる。$ a $ と $ b $ の両方が小さい数であっても、対数値は適切な範囲に収まるはずである。

```
a = 0.1
print('underflow:', a**784)
print('logarithm is normal:', 784*math.log(a))
```

```
underflow: 0.0
logarithm is normal: -1805.2267129073316
```


```
a = 0.1
print('underflow:', a**784)
print('logarithm is normal:', 784*math.log(a))
```

```
underflow: 0.0
logarithm is normal: -1805.2267129073316
```


```
a = 0.1
print('underflow:', a**784)
print('logarithm is normal:', 784*tf.math.log(a).numpy())
```

```
underflow: 0.0
logarithm is normal: -1805.2267379760742
```

対数は単調増加関数なので、(25.9.6) を次のように書き換えられる。

$$\hat{y} = \operatorname{argmax}_y \ \log P_y[y] + \sum_{i=1}^{d} \Big[t_i \log P_{xy}[x_i, y] + (1 - t_i) \log(1 - P_{xy}[x_i, y]) \Big]. \tag{25.9.7}$$

次の安定版を実装できる。

```
log_P_xy = torch.log(P_xy)
log_P_xy_neg = torch.log(1 - P_xy)
log_P_y = torch.log(P_y)

def bayes_pred_stable(x):
    x = x.unsqueeze(0)  # (28, 28) -> (1, 28, 28)
```

(continues on next page)

(continued from previous page)

```
    p_xy = log_P_xy * x + log_P_xy_neg * (1 - x)
    p_xy = p_xy.reshape(10, -1).sum(dim=1)  # p(x|y)
    return p_xy + log_P_y

py = bayes_pred_stable(image)
py
```

```
tensor([-268.9725, -301.7044, -245.1951, -218.8738, -193.4570, -206.0909,
        -292.5226, -114.6257, -220.3313, -163.1784])
```

```
log_P_xy = np.log(P_xy)
log_P_xy_neg = np.log(1 - P_xy)
log_P_y = np.log(P_y)

def bayes_pred_stable(x):
    x = np.expand_dims(x, axis=0)  # (28, 28) -> (1, 28, 28)
    p_xy = log_P_xy * x + log_P_xy_neg * (1 - x)
    p_xy = p_xy.reshape(10, -1).sum(axis=1)  # p(x|y)
    return p_xy + log_P_y

py = bayes_pred_stable(image)
py
```

```
array([-268.97253, -301.7044 , -245.19514, -218.87384, -193.45703,
       -206.09088, -292.52264, -114.62566, -220.33133, -163.17842])
```

```
log_P_xy = tf.math.log(P_xy)
log_P_xy_neg = tf.math.log(1 - P_xy)
log_P_y = tf.math.log(P_y)

def bayes_pred_stable(x):
    x = tf.expand_dims(x, axis=0)  # (28, 28) -> (1, 28, 28)
    p_xy = log_P_xy * x + log_P_xy_neg * (1 - x)
    p_xy = tf.math.reduce_sum(tf.reshape(p_xy, (10, -1)), axis=1)  # p(x|y)
    return p_xy + log_P_y

py = bayes_pred_stable(image)
py
```

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([-266.65015, -320.79282, -261.50186, -202.62967, -295.57236,
       -199.49718, -318.66226, -266.01474, -224.59566, -271.9329 ],
      dtype=float32)>
```

これで予測が正しいかどうか確認できる。

```
py.argmax(dim=0) == label
```

```
tensor(True)
```

```
# Convert label which is a scalar tensor of int32 dtype to a Python scalar
# integer for comparison
py.argmax(axis=0) == int(label)
```

```
array(True)
```

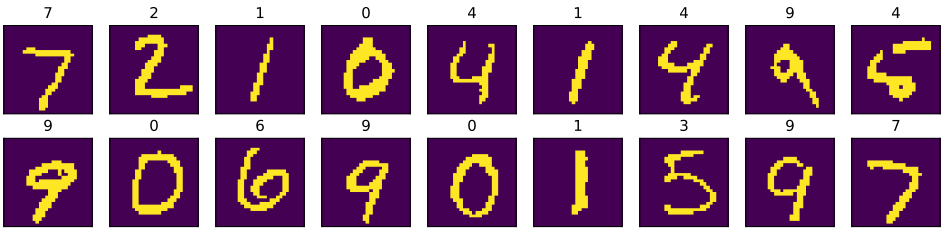
```
tf.argmax(py, axis=0, output_type = tf.int32) == label
```

```
<tf.Tensor: shape=(), dtype=bool, numpy=True>
```

いくつかの検証例を予測してみると、ベイズ分類器がかなりうまく機能していることが分かる。

```
def predict(X):
    return [bayes_pred_stable(x).argmax(dim=0).type(torch.int32).item()
            for x in X]

X = torch.stack([mnist_test[i][0] for i in range(18)], dim=0)
y = torch.tensor([mnist_test[i][1] for i in range(18)])
preds = predict(X)
d2l.show_images(X, 2, 9, titles=[str(d) for d in preds]);
```

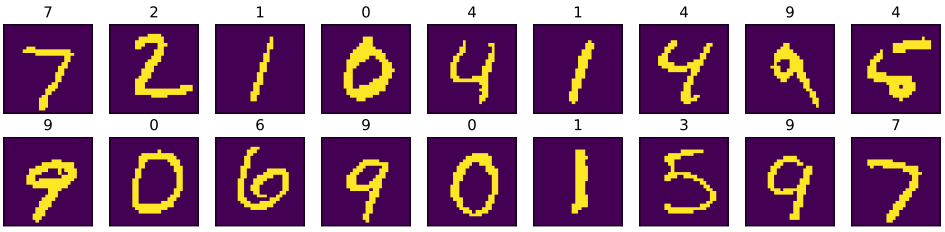



```
def predict(X):
    return [bayes_pred_stable(x).argmax(axis=0).astype(np.int32) for x in X]

X, y = mnist_test[:18]
preds = predict(X)
d2l.show_images(X, 2, 9, titles=[str(d) for d in preds]);
```

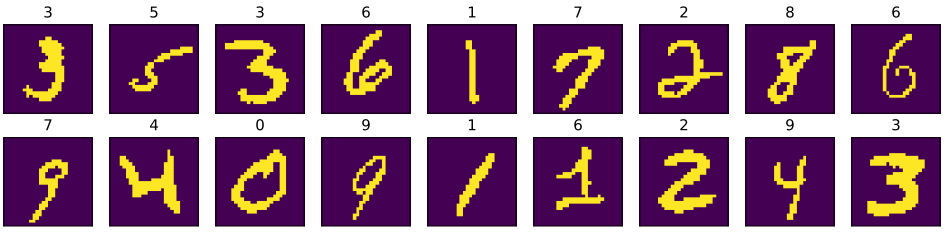
```
def predict(X):
    return [tf.argmax(
        bayes_pred_stable(x), axis=0, output_type = tf.int32).numpy()
            for x in X]
```

(continues on next page)



(continued from previous page)

```
X = tf.stack([train_images[i] for i in range(10, 38)], axis=0)
y = tf.constant([train_labels[i].numpy() for i in range(10, 38)])
preds = predict(X)
d2l.show_images(X, 2, 9, titles=[str(d) for d in preds]);
```



最後に、分類器全体の精度を計算しよう。

```
X = torch.stack([mnist_test[i][0] for i in range(len(mnist_test))], dim=0)
y = torch.tensor([mnist_test[i][1] for i in range(len(mnist_test))])
preds = torch.tensor(predict(X), dtype=torch.int32)
float((preds == y).sum()) / len(y)  # Validation accuracy
```

```
0.8427
```

```
X, y = mnist_test[:]
preds = np.array(predict(X), dtype=np.int32)
float((preds == y).sum()) / len(y)  # Validation accuracy
```

```
0.8427
```

```
X = test_images
y = test_labels
preds = tf.constant(predict(X), dtype=tf.int32)
# Validation accuracy
tf.reduce_sum(tf.cast(preds == y, tf.float32)).numpy() / len(y)
```

```
0.8427
```

現代の深層ネットワークは、0.01 未満の誤り率を達成する。比較的性能が低いのは、モデルで置いた統計的仮定が誤っているからである。つまり、各画素がラベルのみに依存して独立に生成されると仮定したのである。これは人間が数字を書く方法とは明らかに異なり、この誤った仮定が、あまりにもナイーブな（ベイズ）分類器の失敗につながりた。

25.9.5 まとめ

- ベイズの規則を用いると、観測されたすべての特徴が独立であると仮定することで分類器を作れる。
- この分類器は、ラベルと画素値の組合せの出現回数を数えることで、データセット上で学習できる。
- この分類器は、スパム検出のようなタスクにおいて何十年もの間、標準的な手法だった。

25.9.6 演習

1. データセット $[[0,0],[0,1],[1,0],[1,1]]$ を考え、ラベルを 2 つの要素の XOR で与えた $[0,1,1,0]$ とする。このデータセット上に構築したナイーブベイズ分類器の確率はどうなるか。点を正しく分類できるか。できない場合、どの仮定が破られているか。
2. 確率を推定する際にラプラス平滑化を使わず、テスト時に学習で一度も観測されなかった値を含むデータ例が来たとする。このときモデルは何を出力するだろうか。
3. ナイーブベイズ分類器はベイジアンネットワークの特別な例であり、確率変数の依存関係がグラフ構造で表現される。この節の範囲を超える完全な理論については Koller and Friedman (2009) を参照されたいが、XOR モデルにおいて 2 つの入力変数の間の明示的な依存を許すと、なぜ成功する分類器を作れるのか説明する。

25.10 統計学

言うまでもなく、最先端で高精度なモデルを訓練できることは、優れた深層学習実践者になるうえで極めて重要である。しかし、改善が統計的に有意なのか、それとも訓練過程における偶然の変動の結果にすぎないのかは、しばしば明確ではない。推定値における不確実性について議論できるようになるために、私たちは統計学を学ぶ必要がある。

統計学に関する最も古い記録は、9 世紀のアラブの学者アル＝キンディーにさかのぼることができる。彼は、暗号化されたメッセージを解読するために統計と頻度分析をどのように用いるかを詳細に記述した。それから 800 年後、近代統計学は 1700 年代のドイツで生まれた。当時の研究者たちは、人口統計および経済データの収集と分析に注目していた。今日、統計学はデータの収集、処理、分析、解釈、可視化を扱う学問である。さらに、統計学の中核理論は、学術界、産業界、政府における研究で広く用いられている。

より具体的には、統計学は 記述統計と 統計的推論に分けられる。前者は、観測されたデータ集合の特徴を要約し、説明することに焦点を当てる。このデータ集合は 標本と呼ばれる。標本は 母集団から抽出される。母集団とは、私たちの実験対象に関係する類似した個体、項目、または事象の全体集合を指する。記述統計とは対照的に、統計的推論は、標本分布がある程度母集団分布を再現できるという仮定に基づいて、与えられた 標本から母集団の特性をさらに推定する。

「機械学習と統計学の本質的な違いは何か」と疑問に思うかもしれない。根本的には、統計学は推論問題に焦点を当てる。この種の問題には、因果推論のような変数間の関係のモデル化や、A/B テストのようなモデルパラメータの統計的有意性の検定が含まれる。これに対して、機械学習は、各パラメータの機能を明示的にプログラムしたり理解したりすることなく、正確な予測を行うことを重視する。

この節では、3 種類の統計的推論手法、すなわち推定量の評価と比較、仮説検定の実施、信頼区間の構成を紹介する。これらの手法は、与えられた母集団、すなわち真のパラメータ θ の特性を推定するのに役立つ。簡潔さのため、ここでは与えられた母集団の真のパラメータ θ がスカラー値であると仮定する。θ がベクトルやテンソルである場合への拡張は容易なので、ここでは議論を省略する。

25.10.1 推定量の評価と比較

統計学において、推定量とは、真のパラメータ θ を推定するために与えられた標本の関数である。標本 $\{x_1, x_2, \ldots, x_n\}$ を観測した後の θ の推定値を $\hat{\theta}_n = \hat{f}(x_1, \ldots, x_n)$ と書く。

推定量の簡単な例は、Section 25.7 節ですでに見た。ベルヌーイ確率変数からいくつかの標本があるとき、その確率変数が 1 である確率の最尤推定値は、観測された 1 の個数を数え、それを標本数で割ることで得られる。同様に、ある演習では、ガウス分布の平均の最尤推定値が、すべての標本の平均値で与えられることを示すよう求められた。これらの推定量が真のパラメータ値を与えることはほとんどないが、理想的には、標本数が十分に大きければ推定値は真値に近づく。

例として、平均 0、分散 1 のガウス確率変数の真の密度と、そのガウス分布から得られた標本を以下に示す。各点が見えるように、また元の密度との関係がより明確になるように、y 座標を構成している。

```
from d2l import torch as d2l
import torch

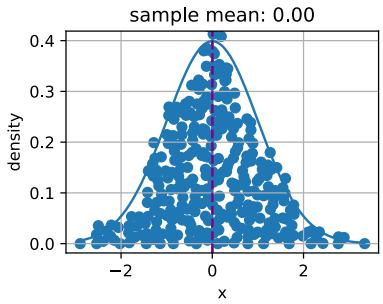
torch.pi = torch.acos(torch.zeros(1)) * 2  #define pi in torch

# Sample datapoints and create y coordinate
epsilon = 0.1
torch.manual_seed(8675309)
xs = torch.randn(size=(300,))

ys = torch.tensor(
    [torch.sum(torch.exp(-(xs[:i] - xs[i])**2 / (2 * epsilon**2))\
               / torch.sqrt(2*torch.pi*epsilon**2)) / len(xs)\
     for i in range(len(xs))])

# Compute true density
xd = torch.arange(torch.min(xs), torch.max(xs), 0.01)
yd = torch.exp(-xd**2/2) / torch.sqrt(2 * torch.pi)

# Plot the results
d2l.plot(xd, yd, 'x', 'density')
d2l.plt.scatter(xs, ys)
d2l.plt.axvline(x=0)
d2l.plt.axvline(x=torch.mean(xs), linestyle='--', color='purple')
d2l.plt.title(f'sample mean: {float(torch.mean(xs).item()):.2f}')
d2l.plt.show()
```




```
from d2l import mxnet as d2l
from mxnet import np, npx
import random
npx.set_np()

# Sample datapoints and create y coordinate
epsilon = 0.1
random.seed(8675309)
xs = np.random.normal(loc=0, scale=1, size=(300,))

ys = [np.sum(np.exp(-(xs[:i] - xs[i])**2 / (2 * epsilon**2))
             / np.sqrt(2*np.pi*epsilon**2)) / len(xs) for i in range(len(xs))]

# Compute true density
```

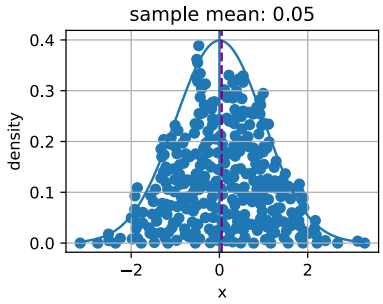
(continues on next page)

(continued from previous page)

```
xd = np.arange(np.min(xs), np.max(xs), 0.01)
yd = np.exp(-xd**2/2) / np.sqrt(2 * np.pi)

# Plot the results
d2l.plot(xd, yd, 'x', 'density')
d2l.plt.scatter(xs, ys)
d2l.plt.axvline(x=0)
d2l.plt.axvline(x=np.mean(xs), linestyle='--', color='purple')
d2l.plt.title(f'sample mean: {float(np.mean(xs)):.2f}')
d2l.plt.show()
```

```
[07:03:44] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```




```
mse(samples, theta_true)
```

```
from d2l import tensorflow as d2l
import tensorflow as tf

tf.pi = tf.acos(tf.zeros(1)) * 2  # define pi in TensorFlow

# Sample datapoints and create y coordinate
epsilon = 0.1
xs = tf.random.normal((300,))

ys = tf.constant(
    [(tf.reduce_sum(tf.exp(-(xs[:i] - xs[i])**2 / (2 * epsilon**2)) \
                / tf.sqrt(2*tf.pi*epsilon**2)) / tf.cast(
           tf.size(xs), dtype=tf.float32)).numpy() \
     for i in range(tf.size(xs))])

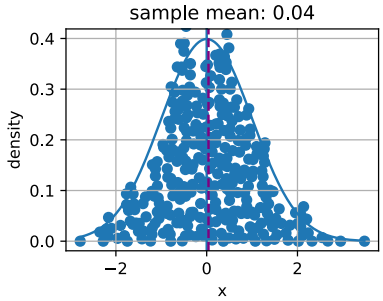
# Compute true density
xd = tf.range(tf.reduce_min(xs), tf.reduce_max(xs), 0.01)
yd = tf.exp(-xd**2/2) / tf.sqrt(2 * tf.pi)

# Plot the results
d2l.plot(xd, yd, 'x', 'density')
d2l.plt.scatter(xs, ys)
```

(continues on next page)

(continued from previous page)

```
d2l.plt.axvline(x=0)
d2l.plt.axvline(x=tf.reduce_mean(xs), linestyle='--', color='purple')
d2l.plt.title(f'sample mean: {float(tf.reduce_mean(xs).numpy()):.2f}')
d2l.plt.show()
```



パラメータ $\hat{\theta}_n$ の推定量を計算する方法は多くある。この節では、推定量を評価・比較するための 3 つの一般的な方法、すなわち平均二乗誤差、標準偏差、統計的バイアスを紹介する。

平均二乗誤差

推定量を評価するために用いられる最も単純な指標は、おそらく 平均二乗誤差（*MSE*）（または l_2 損失）であり、次のように定義できる。

$$\mathrm{MSE}(\hat{\theta}_n, \theta) = E[(\hat{\theta}_n - \theta)^2]. \tag{25.10.1}$$

これにより、真の値からの平均的な二乗偏差を定量化できる。MSE は常に非負である。Section 6.1 を読んでいれば、これが最も一般的に使われる回帰損失関数であることがわかるだろう。推定量を評価する尺度としては、その値が 0 に近いほど、推定量は真のパラメータ θ に近いことを意味する。

統計的バイアス

MSE は自然な指標であるが、それを大きくする要因は複数考えられる。根本的に重要なのは、データセット内のランダム性による推定量の変動と、推定手続きに起因する推定量の系統誤差の 2 つである。

まず、系統誤差を測定しよう。推定量 $\hat{\theta}_n$ に対して、統計的バイアスは数学的には次のように定義できる。

$$\mathrm{bias}(\hat{\theta}_n) = E(\hat{\theta}_n - \theta) = E(\hat{\theta}_n) - \theta. \tag{25.10.2}$$

$\mathrm{bias}(\hat{\theta}_n) = 0$ のとき、推定量 $\hat{\theta}_n$ の期待値はパラメータの真値に等しいことに注意する。この場合、$\hat{\theta}_n$ は不偏推定量であると言う。一般に、不偏推定量は、その期待値が真のパラメータと同じであるため、偏りのある推定量よりも優れている。

ただし、実務では偏りのある推定量が頻繁に使われることに注意する価値がある。追加の仮定なしには不偏推定量が存在しない場合や、計算が困難な場合があるからである。これは推定量の重大な欠点のように見えるかもしれないが、実際に遭遇する推定量の大半は、少なくとも漸近的には不偏である。すなわち、利用可能な標本数が無限大に近づくにつれてバイアスが 0 に近づく：$\lim_{n \to \infty} \text{bias}(\hat{\theta}_n) = 0$。

分散と標準偏差

次に、推定量のランダム性を測定しよう。Section 25.6 で見たように、標準偏差（または 標準誤差）は分散の平方根として定義される。推定量の標準偏差または分散を測ることで、その推定量の変動の程度を評価できる。

$$\sigma_{\hat{\theta}_n} = \sqrt{\text{Var}(\hat{\theta}_n)} = \sqrt{E[(\hat{\theta}_n - E(\hat{\theta}_n))^2]}. \tag{25.10.3}$$

(25.10.3) と (25.10.1) を比較することが重要である。この式では真の母集団値 θ と比較しているのではなく、代わりに $E(\hat{\theta}_n)$、すなわち期待される標本平均と比較している。したがって、推定量が真の値からどれだけ離れがちかを測っているのではなく、推定量そのものの変動を測っているのである。

バイアス・分散トレードオフ

これら 2 つの主要な要素が平均二乗誤差に寄与することは直感的に明らかである。やや驚くべきことに、平均二乗誤差は、これら 2 つの寄与に第 3 の項を加えたものへと実際に 分解できることを示せる。つまり、平均二乗誤差は、バイアスの二乗、分散、そして不可避誤差の和として書ける。

$$\begin{aligned}
\text{MSE}(\hat{\theta}_n, \theta) &= E[(\hat{\theta}_n - \theta)^2] \\
&= E[(\hat{\theta}_n)^2] + E[\theta^2] - 2E[\hat{\theta}_n \theta] \\
&= \text{Var}[\hat{\theta}_n] + E[\hat{\theta}_n]^2 + \text{Var}[\theta] + E[\theta]^2 - 2E[\hat{\theta}_n]E[\theta] \\
&= (E[\hat{\theta}_n] - E[\theta])^2 + \text{Var}[\hat{\theta}_n] + \text{Var}[\theta] \\
&= (E[\hat{\theta}_n - \theta])^2 + \text{Var}[\hat{\theta}_n] + \text{Var}[\theta] \\
&= (\text{bias}[\hat{\theta}_n])^2 + \text{Var}(\hat{\theta}_n) + \text{Var}[\theta].
\end{aligned} \tag{25.10.4}$$

上の式を バイアス・分散トレードオフと呼ぶ。平均二乗誤差は、3 つの誤差源、すなわち高バイアスによる誤差、高分散による誤差、そして不可避誤差に分けられる。バイアス誤差は、特徴量と出力の間の高次元の関係を抽出できない単純なモデル（たとえば線形回帰モデル）でよく見られる。モデルが高バイアス誤差に悩まされるとき、Section 6.6 で述べたように、そのモデルは アンダーフィッティングしている、あるいは 柔軟性が不足していると言う。高分散は通常、訓練データに過剰適合する複雑すぎるモデルから生じる。その結果、過学習したモデルはデータの小さな変動に敏感になる。モデルが高分散に悩まされるとき、Section 6.6 で述べたように、そのモデルは 過学習しており、汎化が不足していると言う。不可避誤差は、θ 自体に含まれるノイズに起因する。

コードによる推定量の評価

推定量の標準偏差は、テンソル a に対して単に a.std() を呼び出すことで実装できるため、ここでは省略し、統計的バイアスと平均二乗誤差を実装する。

```
# Statistical bias
def stat_bias(true_theta, est_theta):
    return(torch.mean(est_theta) - true_theta)

# Mean squared error
def mse(data, true_theta):
    return(torch.mean(torch.square(data - true_theta)))
```

```
# Statistical bias
def stat_bias(true_theta, est_theta):
    return(np.mean(est_theta) - true_theta)

# Mean squared error
def mse(data, true_theta):
    return(np.mean(np.square(data - true_theta)))
```

```
# Statistical bias
def stat_bias(true_theta, est_theta):
    return(tf.reduce_mean(est_theta) - true_theta)

# Mean squared error
def mse(data, true_theta):
    return(tf.reduce_mean(tf.square(data - true_theta)))
```

バイアス・分散トレードオフの式を示すために、正規分布 $\mathcal{N}(\theta, \sigma^2)$ を $10,000$ 個の標本でシミュレーションしてみよう。ここでは $\theta = 1$、$\sigma = 4$ を用いる。推定量は与えられた標本の関数なので、ここではこの正規分布 $\mathcal{N}(\theta, \sigma^2)$ における真の θ の推定量として標本平均を用いる。

```
theta_true = 1
sigma = 4
sample_len = 10000
samples = torch.normal(theta_true, sigma, size=(sample_len, 1))
theta_est = torch.mean(samples)
theta_est
```

```
tensor(1.0170)
```

```
theta_true = 1
sigma = 4
sample_len = 10000
samples = np.random.normal(theta_true, sigma, sample_len)
```

(continues on next page)

(continued from previous page)

```
theta_est = np.mean(samples)
theta_est
```

```
array(0.9503336)
```

```
theta_true = 1
sigma = 4
sample_len = 10000
samples = tf.random.normal((sample_len, 1), theta_true, sigma)
theta_est = tf.reduce_mean(samples)
theta_est
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=1.0038849>
```

推定量のバイアスの二乗と分散の和を計算して、トレードオフの式を検証しよう。まず、推定量の MSE を計算する。

```
mse(samples, theta_true)
```

```
tensor(16.0298)
```

次に、以下のように $\text{Var}(\hat{\theta}_n) + [\text{bias}(\hat{\theta}_n)]^2$ を計算する。見てわかるように、2 つの値は数値精度の範囲で一致する。

```
bias = stat_bias(theta_true, theta_est)
torch.square(samples.std(unbiased=False)) + torch.square(bias)
```

```
tensor(16.0298)
```

```
bias = stat_bias(theta_true, theta_est)
np.square(samples.std()) + np.square(bias)
```

```
array(15.781995)
```

```
bias = stat_bias(theta_true, theta_est)
tf.square(tf.math.reduce_std(samples)) + tf.square(bias)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=15.809735>
```

25.10.2 仮説検定の実施

統計的推論で最もよく遭遇する話題は仮説検定である。仮説検定は 20 世紀初頭に広まったが、その最初の使用は 1700 年代のジョン・アーブスノットにさかのぼることができる。ジョンはロンドンの 80 年間の出生記録を追跡し、毎年、女性よりも男性のほうが多く生まれていると結論づけた。その後、現代的な有意性検定は、p 値とピアソンのカイ二乗検定を考案したカール・ピアソン、スチューデントの t 分布の父であるウィリアム・ゴセット、そして帰無仮説と有意性検定を導入したロナルド・フィッシャーによって発展した。

仮説検定とは、母集団に関する既定の主張に対して、何らかの証拠を評価する方法である。この既定の主張を 帰無仮説 H_0 と呼び、観測データを用いてこれを棄却しようとする。ここでは、H_0 を統計的有意性検定の出発点として用いる。対立仮説 H_A（または H_1）は、帰無仮説に反する主張である。帰無仮説は、変数間の関係を仮定する宣言的な形で述べられることが多いである。それはできるだけ明確に要件を反映し、統計理論によって検証可能でなければならない。

あなたが化学者だと想像する。研究室で何千時間も費やした後、数学を理解する能力を劇的に向上させる新薬を開発したとする。その魔法のような効力を示すには、それを検証する必要がある。当然ながら、何人かのボランティアにその薬を飲んでもらい、数学の学習が本当に良くなるかを確かめる必要があるだろう。では、どう始めればよいだろうか。

まず、数学的理解能力が何らかの指標で測ったときに差がないよう、慎重にランダムに選ばれた 2 つのボランティア群が必要である。この 2 つの群は一般に、テスト群と対照群と呼ばれる。テスト群（または 処置群）は薬を投与される個体群であり、対照群は基準として取っておく群、すなわちこの薬を飲まないこと以外は同一の環境設定に置かれた群を表する。このようにすることで、処置における独立変数の影響を除き、他のすべての変数の影響を最小化できる。

次に、一定期間薬を飲んだ後、同じ指標で 2 つの群の数学的理解度を測定する必要がある。たとえば、新しい数学公式を学んだ後に、ボランティアに同じテストを受けてもらうといった方法である。その後、成績を集めて結果を比較できる。この場合、帰無仮説は 2 つの群に差がないことであり、対立仮説は差があることになる。

しかし、これでもまだ完全に形式化されているわけではない。慎重に考えるべき詳細はたくさんある。たとえば、数学的理解能力を検定するのに適した指標は何だろうか。薬の有効性を主張できると確信するには、何人のボランティアが必要だろうか。検定はどれくらいの期間実施すべきだろうか。2 つの群に差があるかどうかをどう判断すればよいだろうか。平均的な成績だけを気にするのか、それとも得点の変動範囲も気にするのか。などである。

このように、仮説検定は、実験計画と観測結果の確実性について推論するための枠組みを与える。もし帰無仮説が真である可能性が非常に低いことを示せれば、私たちはそれを自信をもって棄却できる。

仮説検定の扱い方を理解するために、ここで追加の用語を導入し、上で述べた概念のいくつかを形式化する必要がある。

統計的有意性

統計的有意性は、棄却すべきでない帰無仮説 H_0 を誤って棄却してしまう確率を表する。すなわち、

$$\text{statistical significance } = 1 - \alpha = 1 - P(\text{reject } H_0 \mid H_0 \text{ is true}). \tag{25.10.5}$$

これは 第 *I* 種の誤りまたは 偽陽性とも呼ばれる。α は 有意水準と呼ばれ、一般的な値は 5%、すなわち $1 - \alpha = 95\%$ である。有意水準は、真の帰無仮説を棄却するときに私たちが受け入れるリスクの大きさとして説明できる。

Fig. 25.10.1 は、2 標本仮説検定における観測値と、与えられた正規分布の確率を示している。観測データの例が 95% の閾値の外側にある場合、それは帰無仮説の下では非常に起こりにくい観測である。したがって、帰無仮説に何か問題がある可能性があり、それを棄却することになる。

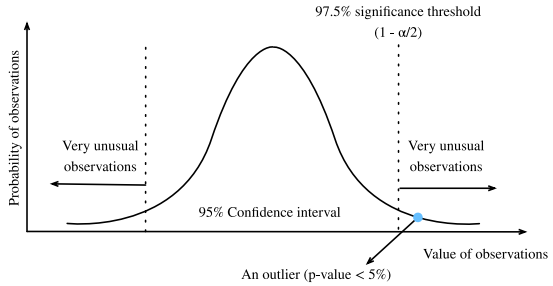


Fig. 25.10.1 Statistical significance.

検出力

統計的検出力（または 感度）は、棄却すべき帰無仮説 H_0 を正しく棄却できる確率を表する。すなわち、

$$\text{statistical power } = 1 - \beta = 1 - P(\text{ fail to reject } H_0 \mid H_0 \text{ is false}). \tag{25.10.6}$$

第 *I* 種の誤りは、真である帰無仮説を棄却してしまうことによる誤りであり、第 *II* 種の誤りは、偽である帰無仮説を棄却できないことによる誤りである。第 II 種の誤りは通常 β で表されるため、対応する統計的検出力は $1 - \beta$ である。

直感的には、統計的検出力は、所望の有意水準のもとで、ある最小の大きさを持つ真の差異をどれだけ検出しやすいかを表していると解釈できる。80% はよく使われる検出力の閾値である。検出力が高いほど、真の差を検出できる可能性が高くなる。

統計的検出力の最も一般的な用途の 1 つは、必要な標本数を決めることであ

る。帰無仮説が偽であるときにそれを棄却できる確率は、それがどの程度偽であるか（効果量と呼ばれます）と、持っている標本数に依存する。予想されるとおり、効果量が小さい場合、それを高い確率で検出するには非常に多くの標本が必要になる。詳細な導出はこの簡潔な付録の範囲を超えるが、例として、標本が平均 0・分散 1 のガウス分布から来たという帰無仮説を棄却したいとし、実際には標本平均が 1 に近いと考えている場合、許容可能な誤差率でそれを行うのに必要な標本サイズはわずか 8 である。しかし、母集団の真の平均が 0.01 に近いと考えるなら、その差を検出するには約 80000 の標本サイズが必要になる。

検出力は水フィルターにたとえることができる。この比喩では、高検出力の仮説検定は、高品質な浄水システムのように、水中の有害物質をできるだけ多く除去する。一方、差異が小さい場合は低品質な水フィルターのようなもので、比較的小さな物質が隙間から容易に漏れ出してしまいる。同様に、統計的検出力が十分に高くないと、検定は小さな差異を見逃してしまうかもしれない。

検定統計量

検定統計量 $T(x)$ は、標本データのある特徴を要約するスカラーである。このような統計量を定義する目的は、異なる分布を区別し、仮説検定を実施できるようにすることである。化学者の例に戻ると、ある母集団が別の母集団より優れていることを示したいなら、平均を検定統計量として取るのは妥当だろう。検定統計量の選び方によって、統計検定の検出力は大きく変わり得る。

しばしば、$T(X)$（帰無仮説の下での検定統計量の分布）は、少なくとも近似的には、帰無仮説のもとで正規分布のような一般的な確率分布に従う。このような分布を明示的に導出でき、さらにデータセット上で検定統計量を測定できれば、その統計量が期待される範囲から大きく外れているときに、帰無仮説を安全に棄却できる。これを定量化したものが p 値の概念である。

p 値

p 値（または 確率値）は、帰無仮説が 真であると仮定したときに、$T(X)$ が観測された検定統計量 $T(x)$ と少なくとも同程度に極端である確率である。すなわち、

$$p\text{-value} = P_{H_0}(T(X) \geq T(x)). \tag{25.10.7}$$

p 値が、あらかじめ定めた固定の統計的有意水準 α 以下であれば、帰無仮説を棄却できる。そうでなければ、帰無仮説を棄却する証拠が不足していると結論づける。与えられた母集団分布に対して、棄却域は、p 値が統計的有意水準 α より小さいすべての点を含む区間になる。

片側検定と両側検定

通常、有意性検定には 2 種類ある。片側検定と両側検定である。片側検定（または 片側仮説検定）は、帰無仮説と対立仮説が一方向にしかない場合に適用される。たとえば、帰無仮説が真のパラメータ θ が値 c 以下であると述べることがある。対立仮説は θ が c より大きいことになる。つまり、棄却域は標本分布の片側にのみある。片側検定とは対照的に、両側検定（または 両側仮説検定）は、棄却域が標本分布の両側にある場合に適用される。この場合の例として、帰無仮説が真のパラメータ θ が値 c に等しいと述べることがある。対立仮説は θ が c に等しくないことになる。

仮説検定の一般的な手順

上の概念に慣れたところで、仮説検定の一般的な手順を見ていきよう。

1. 問題を定義し、帰無仮説 H_0 を立てる。

2. 統計的有意水準 α と統計的検出力 $(1 - \beta)$ を設定する。

3. 実験を通じて標本を得る。必要な標本数は、統計的検出力と期待される効果量に依存する。

4. 検定統計量と p 値を計算する。

5. p 値と統計的有意水準 α に基づいて、帰無仮説を維持するか棄却するかを決定する。

仮説検定を行うには、まず帰無仮説と、私たちが受け入れるリスクの水準を定義する。次に、標本の検定統計量を計算し、その極端な値を帰無仮説に対する証拠として扱う。検定統計量が棄却域に入れば、対立仮説を支持して帰無仮説を棄却できる。

仮説検定は、臨床試験や A/B テストなど、さまざまな場面に適用できる。

25.10.3 信頼区間の構成

パラメータ θ の値を推定するとき、$\hat{\theta}$ のような点推定量は、不確実性の概念を含まないため、限られた有用性しかない。むしろ、真のパラメータ θ を高い確率で含む区間を作れたほうがはるかに望ましいだろう。もし 1 世紀前にそのような考えに関心があったなら、1937 年に信頼区間の概念を初めて導入した Jerzy Neyman の“Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability”(Neyman, 1937) を読んで興奮したことだろう。

有用であるためには、信頼区間は、与えられた確実性の程度に対してできるだけ小さくあるべきである。どのように導出するかを見てみよう。

定義

数学的には、真のパラメータ θ に対する 信頼区間とは、標本データから計算される区間 C_n であり、次を満たすものである。

$$P_\theta(C_n \ni \theta) \geq 1 - \alpha, \forall \theta. \tag{25.10.8}$$

ここで $\alpha \in (0,1)$ であり、$1-\alpha$ は区間の 信頼水準または 被覆率と呼ばれる。これは、上で述べた有意水準と同じ α である。

(25.10.8) は固定された θ についてではなく、変数 C_n について述べていることに注意する。これを強調するために、$P_\theta(C_n \ni \theta)$ と書き、$P_\theta(\theta \in C_n)$ とは書かない。

解釈

95% 信頼区間を「真のパラメータが 95% の確率で入っている区間」と解釈したくなるが、残念ながらこれは正しくない。真のパラメータは固定されており、ランダムなのは区間のほうである。したがって、より適切な解釈は、この手続きで多数の信頼区間を生成したなら、そのうち 95% の区間が真のパラメータを含む、というものである。

これは些細な区別に見えるかもしれないが、結果の解釈に実際の影響を及ぼし得る。特に、めったにそうならない限り、真の値を含まないことが ほぼ確実な区間を構成することで (25.10.8) を満たすこともできてしまう。この節の最後に、魅力的に見えるが誤りである 3 つの主張を挙げる。これらの点についての詳細な議論は Morey *et al.* (2016) を参照する。

- **誤解 1**. 信頼区間が狭いほど、パラメータを精密に推定できる。
- **誤解 2**. 信頼区間の内側の値は、外側の値よりも真の値である可能性が高い。
- **誤解 3**. ある特定の観測された 95% 信頼区間が真の値を含む確率は 95% である。

言うまでもなく、信頼区間は繊細な対象である。しかし、解釈を明確に保てば、強力な道具になる。

ガウス分布の例

最も古典的な例として、平均と分散が未知のガウス分布の平均に対する信頼区間を考えよう。ガウス分布 $\mathcal{N}(\mu, \sigma^2)$ から n 個の標本 $\{x_i\}_{i=1}^n$ を集めたとする。平均と分散の推定量は次のように計算できる。

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^n x_i \text{ and } \hat{\sigma}_n^2 = \frac{1}{n-1}\sum_{i=1}^n (x_i - \hat{\mu})^2. \tag{25.10.9}$$

ここで確率変数

$$T = \frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n/\sqrt{n}}, \tag{25.10.10}$$

を考えると、自由度 $n-1$ の スチューデントの:math:`t` 分布と呼ばれるよく知られた分布に従う確率変数が得られる。

この分布は非常によく研究されており、たとえば $n \to \infty$ のとき、近似的に標準ガウス分布になることが知られている。したがって、ガウス分布の c.d.f. の値を表で調べれば、T の値が少なくとも 95% の確率で区間 $[-1.96, 1.96]$ に入ると結論できる。有限の n では、この区間はやや広くする必要があるが、値はよく知られており、表にあらかじめ計算されている。

したがって、大きな n に対しては、

$$P\left(\frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n/\sqrt{n}} \in [-1.96, 1.96]\right) \geq 0.95. \tag{25.10.11}$$

これを両辺に $\hat{\sigma}_n/\sqrt{n}$ を掛け、その後 $\hat{\mu}_n$ を加えて整理すると、

$$P\left(\mu \in \left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]\right) \geq 0.95. \tag{25.10.12}$$

したがって、95% 信頼区間は次のように得られる。

$$\left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]. \tag{25.10.13}$$

(25.10.13) は統計学で最もよく使われる式の 1 つと言ってよいだろう。では、これを実装して統計学の議論を締めくくろう。簡単のため、ここでは漸近的な状況にあると仮定する。小さい N の場合には、プログラムで、あるいは t 表から得た正しい `t_star` の値を使うべきである。

```
# PyTorch uses Bessel's correction by default, which means the use of ddof=1
# instead of default ddof=0 in numpy. We can use unbiased=False to imitate
# ddof=0.

# Number of samples
N = 1000

# Sample dataset
samples = torch.normal(0, 1, size=(N,))

# Lookup Students's t-distribution c.d.f.
t_star = 1.96

# Construct interval
mu_hat = torch.mean(samples)
sigma_hat = samples.std(unbiased=True)
(mu_hat - t_star*sigma_hat/torch.sqrt(torch.tensor(N, dtype=torch.float32)),\
 mu_hat + t_star*sigma_hat/torch.sqrt(torch.tensor(N, dtype=torch.float32)))
```

```
(tensor(-0.0568), tensor(0.0704))
```

```
# Number of samples
N = 1000
```

(continues on next page)

(continued from previous page)

```
# Sample dataset
samples = np.random.normal(loc=0, scale=1, size=(N,))

# Lookup Students's t-distribution c.d.f.
t_star = 1.96

# Construct interval
mu_hat = np.mean(samples)
sigma_hat = samples.std(ddof=1)
(mu_hat - t_star*sigma_hat/np.sqrt(N), mu_hat + t_star*sigma_hat/np.sqrt(N))
```

```
(array(-0.07853346), array(0.04412608))
```

```
# Number of samples
N = 1000

# Sample dataset
samples = tf.random.normal((N,), 0, 1)

# Lookup Students's t-distribution c.d.f.
t_star = 1.96

# Construct interval
mu_hat = tf.reduce_mean(samples)
sigma_hat = tf.math.reduce_std(samples)
(mu_hat - t_star*sigma_hat/tf.sqrt(tf.constant(N, dtype=tf.float32)), \
 mu_hat + t_star*sigma_hat/tf.sqrt(tf.constant(N, dtype=tf.float32)))
```

```
(<tf.Tensor: shape=(), dtype=float32, numpy=-0.092176765>,
 <tf.Tensor: shape=(), dtype=float32, numpy=0.032888114>)
```

25.10.4 まとめ

- 統計学は推論問題に焦点を当てるのに対し、深層学習は各パラメータを明示的にプログラムしたり理解したりすることなく、正確な予測を行うことを重視する。
- 統計的推論の一般的な手法は 3 つある。推定量の評価と比較、仮説検定の実施、信頼区間の構成である。
- 最も一般的な 3 つの指標は、統計的バイアス、標準偏差、平均二乗誤差である。
- 信頼区間とは、与えられた標本から構成できる、真の母集団パラメータの推定範囲である。
- 仮説検定は、母集団に関する既定の主張に対して証拠を評価する方法である。

25.10.5 演習

1. $X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \text{Unif}(0, \theta)$ とする。ここで“iid”は 独立同分布を意味する。θ の次の推定量を考える。

$$\hat{\theta} = \max\{X_1, X_2, \ldots, X_n\}; \tag{25.10.14}$$

$$\tilde{\theta} = 2\bar{X}_n = \frac{2}{n}\sum_{i=1}^{n} X_i. \tag{25.10.15}$$

   - $\hat{\theta}$ の統計的バイアス、標準偏差、平均二乗誤差を求めよ。
   - $\tilde{\theta}$ の統計的バイアス、標準偏差、平均二乗誤差を求めよ。
   - どちらの推定量がより良いか。

2. 導入部の化学者の例について、両側仮説検定を行うための 5 つの手順を導出できるか。有意水準 $\alpha = 0.05$ と統計的検出力 $1 - \beta = 0.8$ を用いよ。

3. $N = 2$ と $\alpha = 0.5$ で信頼区間のコードを、独立に生成した 100 個のデータセットに対して実行し、得られた区間をプロットせよ（この場合 `t_star = 1.0`）。真の平均 0 を含むにはあまりにも遠い、非常に短い区間がいくつか見えるはずである。これは信頼区間の解釈と矛盾するか。短い区間を高精度な推定の指標として使うことに安心感を持てるか。

25.11 情報理論

宇宙は情報であふれている。情報は、学問分野の壁を越える共通言語を与えてくれる。シェイクスピアのソネットから Cornell ArXiv 上の研究論文まで、ゴッホの《星月夜》の版画からベートーヴェンの交響曲第 5 番まで、最初のプログラミング言語 Plankalkül から最先端の機械学習アルゴリズムまで、あらゆるものは情報理論の法則に従わなければならない。形式が何であれ同じである。情報理論を用いれば、異なる信号にどれだけの情報が含まれているかを測定し、比較できる。この節では、情報理論の基本概念と、機械学習における情報理論の応用を調べる。

始める前に、機械学習と情報理論の関係を概観しておきよう。機械学習は、データから興味深い信号を抽出し、重要な予測を行うことを目指する。一方、情報理論は、情報の符号化、復号、伝送、操作を研究する。その結果、情報理論は、機械学習システムにおける情報処理を議論するための基礎的な言語を提供する。たとえば、多くの機械学習アプリケーションでは、Section 7.1 で説明したクロスエントロピー損失を用いる。この損失は、情報理論的な考察から直接導出できる。

25.11.1 情報

情報理論の「魂」である情報から始めよう。情報は、1 つ以上の符号化形式の特定の並びを持つあらゆるものに符号化できる。ここで、情報という概念を定義しようとしているとしよう。出発点として何が考えられるだろうか。

次の思考実験を考えてみる。友人がトランプのデッキを持っている。友人はデッキをシャッフルし、いくつかのカードをめくって、そのカードについての文を私たちに伝える。私たちは、それぞれの文が持つ情報量を評価しようとする。

まず、友人が 1 枚のカードをめくって「カードが見える」と言う。これはまったく情報を与えない。私たちはすでにそうであると確信していたので、情報量はゼロであるべきである。

次に、友人が 1 枚のカードをめくって「ハートが見える」と言う。これはある程度の情報を与えるが、実際には可能なスートは 4 種類しかなく、それぞれ同じ確率で起こるので、この結果に驚きはない。どのような情報量の尺度であっても、この事象の情報量は低いはずである。

次に、友人が 1 枚のカードをめくって「これはスペードの 3 だ」と言う。これはより多くの情報である。実際、可能な結果は 52 通りあり、それらはすべて同確率で、友人はそのうちのどれであるかを教えてくれた。これは中程度の情報量であるべきである。

これを論理的に極限まで進めてみよう。最後に、友人がデッキのすべてのカードをめくり、シャッフルされたデッキの全順序を読み上げるとする。デッキの並び方は 52! 通りあり、これもすべて同確率なので、それがどれかを知るには大量の情報が必要である。

私たちが構築する情報の概念は、この直感に一致していなければならない。実際、次の節では、これらの事象の情報量がそれぞれ 0 bits、2 bits、 5.7 bits、225.6 bits であることを計算する。

これらの思考実験を読むと、自然な考えが見えてきる。出発点として、知識そのものよりも、情報とは驚きの度合い、あるいは事象の抽象的な起こりやすさを表すものだと考えられる。たとえば、珍しい事象を記述したいなら、多くの情報が必要である。ありふれた事象なら、それほど多くの情報は必要ない。

1948 年、Claude E. Shannon は *A Mathematical Theory of Communication* (Shannon, 1948) を発表し、情報理論を確立した。彼の論文で、Shannon は情報エントロピーの概念を初めて導入した。ここから私たちの旅を始めよう。

自己情報

情報は事象の抽象的な起こりやすさを表すので、その起こりやすさをどのようにビット数へ写像すればよいのだろうか。Shannon は、情報の単位として *bit* という用語を導入した。これはもともと John Tukey によって作られたものであ

る。では、「ビット」とは何で、なぜ情報の測定に使うのだろうか。歴史的には、古い送信機は 0 と 1 の 2 種類の符号しか送受信できなかった。実際、2 進符号化は現在のすべてのデジタル計算機でも広く使われている。このように、あらゆる情報は 0 と 1 の列として符号化される。したがって、長さ n の 2 進数字列は n ビットの情報を含む。

ここで、任意の符号列について、各 0 または 1 が確率 $\frac{1}{2}$ で現れるとする。すると、長さ n の符号列を持つ事象 X は、確率 $\frac{1}{2^n}$ で起こる。同時に、先ほど述べたように、この列は n ビットの情報を含む。では、確率 p をビット数に変換する数学関数へ一般化できるだろうか。Shannon は 自己情報を定義することで答えを与えた。

$$I(X) = -\log_2(p), \tag{25.11.1}$$

これは事象 X について受け取った情報の ビット数である。この節では常に底 2 の対数を使うことに注意する。簡単のため、この節の残りでは対数表記の添字 2 を省略し、すなわち $\log(.)$ は常に $\log_2(.)$ を意味する。たとえば、符号“0010”の自己情報は

$$I(\text{"0010"}) = -\log(p(\text{"0010"})) = -\log\left(\frac{1}{2^4}\right) = 4 \text{ bits}. \tag{25.11.2}$$

以下のように自己情報を計算できる。その前に、この節で必要なパッケージをすべてインポートしよう。

```
import torch
from torch.nn import NLLLoss

def nansum(x):
    # Define nansum, as pytorch does not offer it inbuilt.
    return x[~torch.isnan(x)].sum()

def self_information(p):
    return -torch.log2(torch.tensor(p)).item()

self_information(1 / 64)
```

```
6.0
```

```
from mxnet import np
from mxnet.metric import NegativeLogLikelihood
from mxnet.ndarray import nansum
import random

def self_information(p):
    return -np.log2(p)

self_information(1 / 64)
```

```
6.0
```

```
kl_pq1 = kl_divergence(p, q1)
kl_pq2 = kl_divergence(p, q2)
similar_percentage = abs(kl_pq1 - kl_pq2) / ((kl_pq1 + kl_pq2) / 2) * 100

kl_pq1, kl_pq2, similar_percentage
```

```
import tensorflow as tf

def log2(x):
    return tf.math.log(x) / tf.math.log(2.)

def nansum(x):
    return tf.reduce_sum(tf.where(tf.math.is_nan(
        x), tf.zeros_like(x), x), axis=-1)

def self_information(p):
    return -log2(tf.constant(p)).numpy()

self_information(1 / 64)
```

```
6.0
```

25.11.2 エントロピー

自己情報は単一の離散事象の情報しか測れないので、離散分布でも連続分布でもよい任意の確率変数に対する、より一般化された尺度が必要である。

エントロピーの動機づけ

何を求めたいのかを具体的にしてみよう。これは、*Shannon* エントロピーの公理として知られるものの非形式的な説明である。以下の常識的な主張の集まりが、情報の一意な定義を強いることがわかる。これらの公理の形式的な版や他の公理については、Csiszár (2008) を参照する。

1. 確率変数を観測して得られる情報は、要素の呼び名や、確率 0 の追加要素の有無に依存しない。
2. 2 つの確率変数を観測して得られる情報は、それらを別々に観測して得られる情報の和を超えない。独立であれば、ちょうどその和になる。
3. （ほぼ）確実な事象を観測して得られる情報は（ほぼ）ゼロである。

この事実の証明は本書の範囲を超えるが、エントロピーがどのような形を取るべきかを一意に決めることは重要である。これらが許す曖昧さは基本単位の選択だけであり、通常は、単一の公平なコイン投げが 1 ビットの情報を与えるという先ほどの選択で正規化される。

定義

確率密度関数（p.d.f.）または確率質量関数（p.m.f.）$p(x)$ を持つ確率分布 P に従う任意の確率変数 X に対して、期待される情報量を エントロピー（または *Shannon* エントロピー）で測りる。

$$H(X) = -E_{x \sim P}[\log p(x)]. \tag{25.11.3}$$

具体的には、X が離散的なら、

$$H(X) = -\sum_i p_i \log p_i, \text{ where } p_i = P(X_i). \tag{25.11.4}$$

それ以外に、X が連続的なら、エントロピーは 微分エントロピーとも呼ばれる。

$$H(X) = -\int_x p(x) \log p(x) \, dx. \tag{25.11.5}$$

エントロピーは以下のように定義できる。

```
def entropy(p):
    entropy = - p * torch.log2(p)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(entropy)
    return out

entropy(torch.tensor([0.1, 0.5, 0.1, 0.3]))
```

```
tensor(1.6855)
```

```
def entropy(p):
    entropy = - p * np.log2(p)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(entropy.as_nd_ndarray())
    return out

entropy(np.array([0.1, 0.5, 0.1, 0.3]))
```

```
[07:18:23] ../src/storage/storage.cc:196: Using Pooled (Naive) StorageManager␣
↪for CPU
```

```
[1.6854753]
<NDArray 1 @cpu(0)>
```

```
kl_q2p = kl_divergence(q2, p)
differ_percentage = abs(kl_q2p - kl_pq2) / ((kl_q2p + kl_pq2) / 2) * 100

kl_q2p, differ_percentage
```

```
def entropy(p):
    return nansum(- p * log2(p))

entropy(tf.constant([0.1, 0.5, 0.1, 0.3]))
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=1.6854753>
```

解釈

エントロピーの定義 (25.11.3) で、なぜ負の対数の期待値を使うのか疑問に思うかもしれない。いくつか直感を示する。

まず、なぜ 対数関数 log を使うのだろうか。$p(x) = f_1(x)f_2(x)\ldots, f_n(x)$ であり、各成分関数 $f_i(x)$ が互いに独立だとする。これは、各 $f_i(x)$ が $p(x)$ から得られる総情報量に独立に寄与することを意味する。先ほど述べたように、エントロピーの式は独立な確率変数に対して加法的であることが望まれる。幸い、log は確率分布の積を自然に各項の和へ変換できる。

次に、なぜ 負の log を使うのだろうか。直感的には、ありふれた事象は珍しい事象よりも少ない情報しか含まないはずである。なぜなら、普通の事象よりも異例の事象から多くの情報を得ることが多いからである。しかし、log は確率に対して単調増加であり、しかも $[0, 1]$ のすべての値で負である。事象の確率とそのエントロピーの間に単調減少の関係を作る必要があり、しかもそれは理想的には常に正であるべきです（観測したものが、これまで知っていたことを忘れさせるようなことはないはずだからです）。そこで、log 関数の前に負号を付ける。

最後に、期待値関数はどこから来るのだろうか。確率変数 X を考える。自己情報（$-\log(p)$）は、特定の結果を見たときの 驚きの量と解釈できる。実際、確率が 0 に近づくと驚きは無限大になる。同様に、エントロピーは X を観測したときの平均的な驚きの量と解釈できる。たとえば、スロットマシンのシステムが、確率 $p_1, \ldots, p_k$ で統計的に独立な記号 $s_1, \ldots, s_k$ を出力するとする。このとき、このシステムのエントロピーは、各出力を観測したときの自己情報の平均に等しく、すなわち

$$H(S) = \sum_i p_i \cdot I(s_i) = -\sum_i p_i \cdot \log p_i. \tag{25.11.6}$$

エントロピーの性質

上の例と解釈から、エントロピー (25.11.3) の次の性質を導ける。ここでは、X を事象、P を X の確率分布と呼ぶ。

- すべての離散的な X について $H(X) \geq 0$（連続的な X ではエントロピーが負になりうる）。
- $X \sim P$ が p.d.f. または p.m.f. $p(x)$ に従い、P を新しい確率分布 Q の p.d.f. また

は p.m.f. $q(x)$ で推定しようとするとき、

$$H(X) = -E_{x\sim P}[\log p(x)] \leq -E_{x\sim P}[\log q(x)], \text{ with equality if and only if } P = Q. \tag{25.11.7}$$

あるいは、$H(X)$ は P から引いた記号を符号化するのに必要な平均ビット数の下限を与える。

- $X \sim P$ のとき、x はすべての可能な結果に均等に広がっている場合に最大の情報量を伝える。具体的には、確率分布 P が k クラス $\{p_1, \ldots, p_k\}$ を持つ離散分布なら、

$$H(X) \leq \log(k), \text{ with equality if and only if } p_i = \frac{1}{k}, \forall i. \tag{25.11.8}$$

P が連続確率変数なら、話はもっと複雑になる。しかし、さらに P が有限区間（すべての値が 0 と 1 の間）に台を持つと仮定すれば、その区間上の一様分布であるときに P は最大エントロピーを持つ。

25.11.3 相互情報量

これまで、単一の確率変数 X のエントロピーを定義したが、確率変数の組 (X, Y) のエントロピーはどうだろうか。これらの手法は、次のような問いに答えようとしていると考えられる。「X と Y を合わせたときに含まれる情報は、それぞれを別々に見た場合と比べて何か。冗長な情報があるのか、それともすべて独自の情報なのか。」

以下の議論では、常に (X, Y) を、p.d.f. または p.m.f. $p_{X,Y}(x, y)$ を持つ同時確率分布 P に従う確率変数の組とし、X と Y はそれぞれ確率分布 $p_X(x)$ と $p_Y(y)$ に従うものとする。

同時エントロピー

単一の確率変数のエントロピー (25.11.3) と同様に、確率変数の組 (X, Y) の 同時エントロピー $H(X, Y)$ を

$$H(X, Y) = -E_{(x,y)\sim P}[\log p_{X,Y}(x, y)]. \tag{25.11.9}$$

と定義する。

厳密には、(X, Y) が離散確率変数の組なら、

$$H(X, Y) = -\sum_x \sum_y p_{X,Y}(x, y) \log p_{X,Y}(x, y). \tag{25.11.10}$$

一方、(X, Y) が連続確率変数の組なら、微分同時エントロピーを次のように定義する。

$$H(X, Y) = -\int_{x,y} p_{X,Y}(x, y) \ \log p_{X,Y}(x, y) \ dx \ dy. \tag{25.11.11}$$

(25.11.9) は、確率変数の組に含まれる総ランダム性を表していると考えられる。

2 つの極端な例として、$X = Y$ が同一の確率変数なら、その組に含まれる情報は 1 つ分とまったく同じであり、$H(X,Y) = H(X) = H(Y)$ である。反対に、X と Y が独立なら、$H(X,Y) = H(X) + H(Y)$ である。実際、確率変数の組に含まれる情報は、どちらか一方のエントロピーより小さくはなく、両方の和より大きくもない。

$$H(X), H(Y) \leq H(X,Y) \leq H(X) + H(Y). \tag{25.11.12}$$

同時エントロピーをゼロから実装してみよう。

```
def joint_entropy(p_xy):
    joint_ent = -p_xy * torch.log2(p_xy)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(joint_ent)
    return out

joint_entropy(torch.tensor([[0.1, 0.5], [0.1, 0.3]]))
```

```
tensor(1.6855)
```

```
def joint_entropy(p_xy):
    joint_ent = -p_xy * np.log2(p_xy)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(joint_ent.as_nd_ndarray())
    return out

joint_entropy(np.array([[0.1, 0.5], [0.1, 0.3]]))
```

```
[1.6854753]
<NDArray 1 @cpu(0)>
```

```
def joint_entropy(p_xy):
    joint_ent = -p_xy * log2(p_xy)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(joint_ent)
    return out

joint_entropy(tf.constant([[0.1, 0.5], [0.1, 0.3]]))
```

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.8321928, 0.8532826],␣
↪dtype=float32)>
```

これは先ほどと同じ コードであるが、今度は 2 つの確率変数の同時分布に対して働くものとして解釈する。

条件付きエントロピー

上で定義した同時エントロピーは、確率変数の組に含まれる情報量を表する。これは有用であるが、しばしば私たちが知りたいものではない。機械学習の設定を考えてみよう。X を画像の画素値を表す確率変数（または確率変数ベクトル）、Y をクラスラベルである確率変数とする。X にはかなりの情報が含まれているはずである。自然画像は複雑だからである。しかし、画像が与えられた後の Y に含まれる情報は少ないはずである。実際、数字の画像なら、その数字が判読不能でない限り、どの数字かの情報はすでに画像の中に含まれているはずである。したがって、情報理論の語彙をさらに広げるには、別の確率変数に条件づけられた確率変数の情報量を扱える必要がある。

確率論では、変数間の関係を測るために 条件付き確率の定義を見た。ここでは同様に、条件付きエントロピー $H(Y \mid X)$ を定義したいと思いる。これは

$$H(Y \mid X) = -E_{(x,y)\sim P}[\log p(y \mid x)], \tag{25.11.13}$$

と書ける。ここで $p(y \mid x) = \frac{p_{X,Y}(x,y)}{p_X(x)}$ は条件付き確率である。具体的には、(X, Y) が離散確率変数の組なら、

$$H(Y \mid X) = -\sum_x \sum_y p(x,y) \log p(y \mid x). \tag{25.11.14}$$

(X, Y) が連続確率変数の組なら、微分条件付きエントロピーも同様に

$$H(Y \mid X) = -\int_x \int_y p(x,y)\ \log p(y \mid x)\ dx\ dy. \tag{25.11.15}$$

と定義される。

ここで自然に、条件付きエントロピー $H(Y \mid X)$ はエントロピー $H(X)$ や同時エントロピー $H(X,Y)$ とどう関係するのか、という疑問が生じる。上の定義を使うと、次のように簡潔に表せる。

$$H(Y \mid X) = H(X,Y) - H(X). \tag{25.11.16}$$

これは直感的に解釈できる。X が与えられたときの Y の情報量 ($H(Y \mid X)$) は、X と Y を合わせた情報量 ($H(X,Y)$) から、すでに X に含まれている情報を引いたものと同じである。これにより、X にも表れている情報を除いた Y の情報が得られる。

では、条件付きエントロピー (25.11.13) をゼロから実装してみよう。

```
def conditional_entropy(p_xy, p_x):
    p_y_given_x = p_xy/p_x
    cond_ent = -p_xy * torch.log2(p_y_given_x)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(cond_ent)
    return out

conditional_entropy(torch.tensor([[0.1, 0.5], [0.2, 0.3]]),
                    torch.tensor([0.2, 0.8]))
```

```
tensor(0.8635)
```

```
def conditional_entropy(p_xy, p_x):
    p_y_given_x = p_xy/p_x
    cond_ent = -p_xy * np.log2(p_y_given_x)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(cond_ent.as_nd_ndarray())
    return out

conditional_entropy(np.array([[0.1, 0.5], [0.2, 0.3]]), np.array([0.2, 0.8]))
```

```
[0.8635472]
<NDArray 1 @cpu(0)>
```

```
def conditional_entropy(p_xy, p_x):
    p_y_given_x = p_xy/p_x
    cond_ent = -p_xy * log2(p_y_given_x)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(cond_ent)
    return out

conditional_entropy(tf.constant([[0.1, 0.5], [0.2, 0.3]]),
                    tf.constant([0.2, 0.8]))
```

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.43903595, 0.42451128],␣
↪dtype=float32)>
```

相互情報量

前節の確率変数 (X,Y) の設定を踏まえると、「Y に含まれるが X には含まれない情報量がわかった今、X と Y の間で共有されている情報量も同様に問えるのだろうか」と思うかもしれない。答えは (X,Y) の 相互情報量であり、$I(X,Y)$ と書く。

形式的な定義にすぐ入るのではなく、まずはこれまで構築した用語だけを使って相互情報量の式を導いてみることで、直感を養いよう。私たちは 2 つの確率変数の間で共有される情報を求めたいのである。これを行う 1 つの方法は、X と Y を合わせたすべての情報から始めて、共有されていない部分を取り除くことである。X と Y を合わせた情報は $H(X,Y)$ で表される。ここから、Y にはなく X にある情報と、X にはなく Y にある情報を引く必要がある。前節で見たように、これはそれぞれ $H(X \mid Y)$ と $H(Y \mid X)$ で与えられる。したがって、相互情報量は

$$I(X,Y) = H(X,Y) - H(Y \mid X) - H(X \mid Y). \tag{25.11.17}$$

実際、これは相互情報量の有効な定義である。これらの項の定義を展開して

まとめると、少し代数計算をするだけで、これは次と同じであることがわかる。

$$I(X,Y) = E_x E_y \left\{ p_{X,Y}(x,y) \log \frac{p_{X,Y}(x,y)}{p_X(x)p_Y(y)} \right\}. \tag{25.11.18}$$

これらの関係はすべて、図 Fig. 25.11.1 にまとめられる。次の主張がすべて $I(X,Y)$ と等価である理由を考えるのは、直感を試すよい練習である。

- $H(X) - H(X \mid Y)$
- $H(Y) - H(Y \mid X)$
- $H(X) + H(Y) - H(X,Y)$

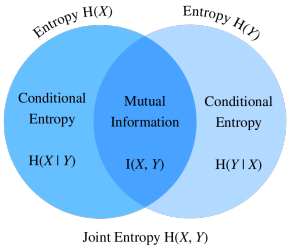


Fig. 25.11.1 相互情報量と同時エントロピー、条件付きエントロピーの関係。

多くの点で、相互情報量 (25.11.18) は、Section 25.6 で見た相関係数を原理的に拡張したものと考えられる。これにより、変数間の線形関係だけでなく、あらゆる種類の 2 つの確率変数が共有する最大の情報量を問うことができる。

では、相互情報量をゼロから実装してみよう。

```
def mutual_information(p_xy, p_x, p_y):
    p = p_xy / (p_x * p_y)
    mutual = p_xy * torch.log2(p)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(mutual)
    return out

mutual_information(torch.tensor([[0.1, 0.5], [0.1, 0.3]]),
                   torch.tensor([0.2, 0.8]), torch.tensor([[0.75, 0.25]]))
```

```
tensor(0.7195)
```

```
def mutual_information(p_xy, p_x, p_y):
    p = p_xy / (p_x * p_y)
    mutual = p_xy * np.log2(p)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(mutual.as_nd_ndarray())
    return out
```

(continues on next page)

(continued from previous page)

```
mutual_information(np.array([[0.1, 0.5], [0.1, 0.3]]),
                   np.array([0.2, 0.8]), np.array([[0.75, 0.25]]))
```

```
[0.7194603]
<NDArray 1 @cpu(0)>
```

```
def mutual_information(p_xy, p_x, p_y):
    p = p_xy / (p_x * p_y)
    mutual = p_xy * log2(p)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(mutual)
    return out

mutual_information(tf.constant([[0.1, 0.5], [0.1, 0.3]]),
                   tf.constant([0.2, 0.8]), tf.constant([[0.75, 0.25]]))
```

```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.60246783, 0.1169925 ],␣
↪dtype=float32)>
```

相互情報量の性質

相互情報量 (25.11.18) の定義を暗記するよりも、次の重要な性質を覚えておけば十分である。

- 相互情報量は対称的である。すなわち、$I(X,Y) = I(Y,X)$。
- 相互情報量は非負である。すなわち、$I(X,Y) \geq 0$。
- X と Y が独立であるとき、かつそのときに限り $I(X,Y) = 0$。たとえば、X と Y が独立なら、Y を知っても X について何の情報も得られず、その逆も同様なので、相互情報量はゼロである。
- あるいは、X が Y の可逆関数であるなら、Y と X はすべての情報を共有しており、

$$I(X,Y) = H(Y) = H(X). \tag{25.11.19}$$

点ごとの相互情報量

この章の冒頭でエントロピーを扱ったとき、$-\log(p_X(x))$ を特定の結果に対する驚きとして解釈できた。相互情報量の対数項にも同様の解釈を与えられる。これはしばしば 点ごとの相互情報量と呼ばれる。

$$\operatorname{pmi}(x,y) = \log \frac{p_{X,Y}(x,y)}{p_X(x)p_Y(y)}. \tag{25.11.20}$$

(25.11.20) は、独立な確率的結果に対して期待される場合と比べて、特定の結果の組 x と y がどれだけ起こりやすいか、あるいは起こりにくいかを測っていると考えられる。大きく正なら、これら 2 つの特定の結果は、ランダムな偶然と比べてはるかに高い頻度で起こります（注：分母は $p_X(x)p_Y(y)$ であり、これは 2 つの結果が独立である場合の確率です）。一方、大きく負なら、ランダムな偶然から期待されるよりもはるかに低い頻度で 2 つの結果が起こることを表する。

これにより、相互情報量 (25.11.18) を、2 つの結果が一緒に起こるのを見たときの驚きの平均量として、独立であると仮定した場合に期待するものと比較して解釈できる。

相互情報量の応用

相互情報量は純粋な定義だけでは少し抽象的であるが、機械学習とどう関係するのだろうか。自然言語処理では、最も難しい問題の 1 つが 曖昧性解消、つまり文脈から単語の意味が不明瞭になる問題である。たとえば、最近のニュース見出しで「Amazon is on fire」と報じられたとする。これは、企業 Amazon の建物が火事なのか、それともアマゾン熱帯雨林が燃えているのか、迷うかもしれない。

この場合、相互情報量はこの曖昧性の解消に役立つ。まず、企業 Amazon と比較的大きな相互情報量を持つ単語群、たとえば e-commerce、technology、online を見つける。次に、アマゾン熱帯雨林と比較的大きな相互情報量を持つ別の単語群、たとえば rain、forest、tropical を見つける。「Amazon」を曖昧でなくする必要があるときは、Amazon という単語の文脈にどちらのグループがより多く現れるかを比較できる。この場合、記事は森林について述べる方向に進み、文脈が明確になる。

25.11.4 Kullback–Leibler ダイバージェンス

Section 5.3 で述べたように、ノルムを使えば任意の次元の空間における 2 点間の距離を測れる。これと同様のことを確率分布に対して行いたいのである。やり方はいくつもあるが、情報理論はその中でも最も優れた方法の 1 つを提供する。ここでは、2 つの分布がどれだけ近いかを測る方法を与える *Kullback–Leibler*（*KL*）ダイバージェンスを調べる。

定義

p.d.f. または p.m.f. $p(x)$ を持つ確率分布 P に従う確率変数 X があり、別の p.d.f. または p.m.f. $q(x)$ を持つ確率分布 Q で P を推定するとする。このとき、P と Q の間の *Kullback–Leibler*（*KL*）ダイバージェンス（または 相対エントロピー）は

$$D_{\mathrm{KL}}(P\|Q) = E_{x\sim P}\left[\log \frac{p(x)}{q(x)}\right]. \tag{25.11.21}$$

点ごとの相互情報量 (25.11.20) と同様に、対数項にも解釈を与えられる。$-\log \frac{q(x)}{p(x)} = -\log(q(x)) - (-\log(p(x)))$ は、P の下で x を見る頻度が Q で期待するよりはるかに高ければ大きく正になり、期待よりはるかに低ければ大きく負になる。このように、これは基準分布から見たときに、その結果を観測してどれだけ 相対的に驚くかを表していると解釈できる。

KL ダイバージェンスをゼロから実装してみよう。

```
def kl_divergence(p, q):
    kl = p * torch.log2(p / q)
    out = nansum(kl)
    return out.abs().item()
```

```
def kl_divergence(p, q):
    kl = p * np.log2(p / q)
    out = nansum(kl.as_nd_ndarray())
    return out.abs().asscalar()
```

```
def kl_divergence(p, q):
    kl = p * log2(p / q)
    out = nansum(kl)
    return tf.abs(out).numpy()
```

KL ダイバージェンスの性質

KL ダイバージェンス (25.11.21) のいくつかの性質を見てみよう。

- KL ダイバージェンスは非対称である。すなわち、ある P, Q について

$$D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P). \tag{25.11.22}$$

- KL ダイバージェンスは非負である。すなわち、

$$D_{\mathrm{KL}}(P\|Q) \geq 0. \tag{25.11.23}$$

  等号が成り立つのは $P = Q$ のときに限られる。

- ある x について $p(x) > 0$ かつ $q(x) = 0$ なら、$D_{\mathrm{KL}}(P\|Q) = \infty$。

- KL ダイバージェンスと相互情報量には密接な関係がある。Fig. 25.11.1 に示した関係に加えて、$I(X, Y)$ は次の項とも数値的に等価である。

  1. $D_{\mathrm{KL}}(P(X, Y) \parallel P(X)P(Y))$;
  2. $E_Y\{D_{\mathrm{KL}}(P(X \mid Y) \parallel P(X))\}$;
  3. $E_X\{D_{\mathrm{KL}}(P(Y \mid X) \parallel P(Y))\}$。

  最初の項では、相互情報量を $P(X, Y)$ と $P(X)$ と $P(Y)$ の積との KL ダイバージェンスとして解釈し、同時分布が独立である場合の分布とどれだけ異な

るかを測るものとみなする。2 番目の項では、X の分布の値を知ることで Y についての不確実性が平均的にどれだけ減るかを相互情報量が教えてくれる。3 番目も同様である。

例

非対称性を明示的に見るために、簡単な例を見てみよう。

まず、長さ $10,000$ の 3 つのテンソルを生成して並べ替える。目的のテンソル p は正規分布 $N(0,1)$ に従い、2 つの候補テンソル q_1 と q_2 はそれぞれ正規分布 $N(-1,1)$ と $N(1,1)$ に従う。

```
torch.manual_seed(1)

tensor_len = 10000
p = torch.normal(0, 1, (tensor_len, ))
q1 = torch.normal(-1, 1, (tensor_len, ))
q2 = torch.normal(1, 1, (tensor_len, ))

p = torch.sort(p)[0]
q1 = torch.sort(q1)[0]
q2 = torch.sort(q2)[0]
```

```
random.seed(1)

nd_len = 10000
p = np.random.normal(loc=0, scale=1, size=(nd_len, ))
q1 = np.random.normal(loc=-1, scale=1, size=(nd_len, ))
q2 = np.random.normal(loc=1, scale=1, size=(nd_len, ))

p = np.array(sorted(p.asnumpy()))
q1 = np.array(sorted(q1.asnumpy()))
q2 = np.array(sorted(q2.asnumpy()))
```

```
tensor_len = 10000
p = tf.random.normal((tensor_len, ), 0, 1)
q1 = tf.random.normal((tensor_len, ), -1, 1)
q2 = tf.random.normal((tensor_len, ), 1, 1)

p = tf.sort(p)
q1 = tf.sort(q1)
q2 = tf.sort(q2)
```

q_1 と q_2 は y 軸（すなわち $x = 0$）に関して対称なので、$D_{\mathrm{KL}}(p\|q_1)$ と $D_{\mathrm{KL}}(p\|q_2)$ は似た値になると期待できる。以下に示すように、$D_{\mathrm{KL}}(p\|q_1)$ と $D_{\mathrm{KL}}(p\|q_2)$ の差は 3% 未満である。

```
kl_pq1 = kl_divergence(p, q1)
kl_pq2 = kl_divergence(p, q2)
similar_percentage = abs(kl_pq1 - kl_pq2) / ((kl_pq1 + kl_pq2) / 2) * 100
```

(continues on next page)

(continued from previous page)

```
kl_pq1, kl_pq2, similar_percentage
```

```
(8582.0341796875, 8828.3095703125, 2.8290698237936858)
```

対照的に、$D_{\mathrm{KL}}(q_2 \| p)$ と $D_{\mathrm{KL}}(p \| q_2)$ は大きく異なり、以下に示すように約 40% の差がある。

```
kl_q2p = kl_divergence(q2, p)
differ_percentage = abs(kl_q2p - kl_pq2) / ((kl_q2p + kl_pq2) / 2) * 100

kl_q2p, differ_percentage
```

```
(14130.125, 46.18621024399691)
```

25.11.5 クロスエントロピー

情報理論の深層学習への応用に興味があるなら、簡単な例を見てみよう。真の分布 P を確率分布 $p(x)$ で、推定分布 Q を確率分布 $q(x)$ で定義し、この節の残りでそれらを使う。

与えられた n 個のデータ例 $\{x_1, \ldots, x_n\}$ に基づいて 2 値分類問題を解く必要があるとする。1 と 0 をそれぞれ正例と負例のラベル y_i として符号化し、ニューラルネットワークが θ でパラメータ化されているとする。$\hat{y}_i = p_\theta(y_i \mid x_i)$ となるような最良の θ を見つけたいなら、Section 25.7 で見た最尤法を適用するのが自然である。具体的には、真のラベル y_i と予測 $\hat{y}_i = p_\theta(y_i \mid x_i)$ に対して、正例に分類される確率は $\pi_i = p_\theta(y_i = 1 \mid x_i)$ である。したがって、対数尤度関数は

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^{n} \pi_i^{y_i} (1 - \pi_i)^{1 - y_i} \\ &= \sum_{i=1}^{n} y_i \log(\pi_i) + (1 - y_i) \log(1 - \pi_i). \end{aligned} \tag{25.11.24}$$

対数尤度関数 $l(\theta)$ を最大化することは、$-l(\theta)$ を最小化することと同じなので、ここから最良の θ を求められる。上の損失を任意の分布へ一般化するために、$-l(\theta)$ を クロスエントロピー損失 $\mathrm{CE}(y, \hat{y})$ とも呼ぶ。ここで y は真の分布 P に従い、$\hat{y}$ は推定分布 Q に従う。

これはすべて最尤推定の観点から導かれた。しかし、よく見ると $\log(\pi_i)$ のような項が計算に現れており、これは情報理論の観点からこの式を理解できることを示す強い示唆である。

形式的定義

KL ダイバージェンスと同様に、確率変数 X に対して、クロスエントロピーによって推定分布 Q と真の分布 P の間のダイバージェンスを測れる。

$$\mathrm{CE}(P, Q) = -E_{x \sim P}[\log(q(x))]. \tag{25.11.25}$$

上で述べたエントロピーの性質を使うと、これはエントロピー $H(P)$ と P と Q の間の KL ダイバージェンスの和としても解釈できる。すなわち、

$$\mathrm{CE}(P, Q) = H(P) + D_{\mathrm{KL}}(P \| Q). \tag{25.11.26}$$

クロスエントロピー損失は以下のように実装できる。

```
def cross_entropy(y_hat, y):
    ce = -torch.log(y_hat[range(len(y_hat)), y])
    return ce.mean()
```

```
def cross_entropy(y_hat, y):
    ce = -np.log(y_hat[range(len(y_hat)), y])
    return ce.mean()
```

```
def cross_entropy(y_hat, y):
    # `tf.gather_nd` is used to select specific indices of a tensor.
    ce = -tf.math.log(tf.gather_nd(y_hat, indices = [[i, j] for i, j in zip(
        range(len(y_hat)), y)]))
    return tf.reduce_mean(ce).numpy()
```

では、ラベルと予測の 2 つのテンソルを定義し、それらのクロスエントロピー損失を計算してみよう。

```
labels = torch.tensor([0, 2])
preds = torch.tensor([[0.3, 0.6, 0.1], [0.2, 0.3, 0.5]])

cross_entropy(preds, labels)
```

```
tensor(0.9486)
```

```
labels = np.array([0, 2])
preds = np.array([[0.3, 0.6, 0.1], [0.2, 0.3, 0.5]])

cross_entropy(preds, labels)
```

```
array(0.94856)
```

```
labels = tf.constant([0, 2])
preds = tf.constant([[0.3, 0.6, 0.1], [0.2, 0.3, 0.5]])

cross_entropy(preds, labels)
```

```
0.94856
```

性質

この節の冒頭で示唆したように、クロスエントロピー (25.11.25) は最適化問題における損失関数として使える。次の 3 つは同値である。

1. 分布 P に対する Q の予測確率を最大化すること、（すなわち $E_{x\sim P}[\log(q(x))]$）;
2. クロスエントロピー $\mathrm{CE}(P, Q)$ を最小化すること;
3. KL ダイバージェンス $D_{\mathrm{KL}}(P\|Q)$ を最小化すること。

クロスエントロピーの定義は、真のデータのエントロピー $H(P)$ が定数である限り、目的 2 と目的 3 の等価関係を間接的に示している。

多クラス分類の目的関数としてのクロスエントロピー

クロスエントロピー損失 CE を用いた分類の目的関数を深く掘り下げると、CE を最小化することは対数尤度関数 L を最大化することと等価であることがわかる。

まず、n 個の例を含むデータセットが与えられ、それが k クラスに分類できるとする。各データ例 i について、任意の k クラスラベル $\mathbf{y}_i = (y_{i1}, \ldots, y_{ik})$ を *one-hot* 符号化で表する。具体的には、例 i がクラス j に属するなら、j 番目の要素を 1 にし、他の成分をすべて 0 にする。すなわち、

$$y_{ij} = \begin{cases} 1 & j \in J; \\ 0 & \text{otherwise.} \end{cases} \tag{25.11.27}$$

たとえば、多クラス分類問題に 3 つのクラス A、B、C があるなら、ラベル $\mathbf{y}_i$ は $\{A : (1, 0, 0); B : (0, 1, 0); C : (0, 0, 1)\}$ と符号化できる。

ニューラルネットワークが θ でパラメータ化されているとする。真のラベルベクトル $\mathbf{y}_i$ と予測

$$\hat{\mathbf{y}}_i = p_\theta(\mathbf{y}_i \mid \mathbf{x}_i) = \sum_{j=1}^{k} y_{ij} p_\theta(y_{ij} \mid \mathbf{x}_i). \tag{25.11.28}$$

したがって、クロスエントロピー損失は

$$\mathrm{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} \mathbf{y}_i \log \hat{\mathbf{y}}_i = -\sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} \log p_\theta(y_{ij} \mid \mathbf{x}_i). \tag{25.11.29}$$

一方で、最尤推定によってもこの問題に取り組める。まず、k クラスの multinoulli 分布を簡単に紹介しよう。これは Bernoulli 分布を 2 値分類から多クラスへ拡張したものである。確率変数 $\mathbf{z} = (z_1, \ldots, z_k)$ が確率 $\mathbf{p} = (p_1, \ldots, p_k)$ を持つ k クラス *multinoulli* 分布に従う、すなわち

$$p(\mathbf{z}) = p(z_1, \ldots, z_k) = \text{Multi}(p_1, \ldots, p_k), \text{ where } \sum_{i=1}^{k} p_i = 1, \tag{25.11.30}$$

とすると、$\mathbf{z}$ の同時確率質量関数 (p.m.f.) は

$$\mathbf{p}^{\mathbf{z}} = \prod_{j=1}^{k} p_j^{z_j}. \tag{25.11.31}$$

各データ例のラベル $\mathbf{y}_i$ は、確率 $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_k)$ を持つ k クラス multinoulli 分布に従うとみなせる。したがって、各データ例 $\mathbf{y}_i$ の同時 p.m.f. は $ß^{\mathbf{y}_i} = \prod_{j=1}^{k} \pi_j^{y_{ij}}$. したがって、対数尤度関数は

$$l(\theta) = \log L(\theta) = \log \prod_{i=1}^{n} \boldsymbol{\pi}^{\mathbf{y}_i} = \log \prod_{i=1}^{n} \prod_{j=1}^{k} \pi_j^{y_{ij}} = \sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} \log \pi_j. \tag{25.11.32}$$

最尤推定では、$\pi_j = p_\theta(y_{ij} \mid \mathbf{x}_i)$ として目的関数 $l(\theta)$ を最大化する。したがって、任意の多クラス分類に対して、上の対数尤度関数 $l(\theta)$ を最大化することは、CE 損失 $\text{CE}(y, \hat{y})$ を最小化することと等価である。

上の証明を確かめるために、組み込みの尺度 `NegativeLogLikelihood` を適用してみよう。先ほどの例と同じ `labels` と `preds` を使うと、前の例と同じ数値損失が小数点以下 5 桁まで一致して得られる。

```
# Implementation of cross-entropy loss in PyTorch combines `nn.LogSoftmax()`
# and `nn.NLLLoss()`
nll_loss = NLLLoss()
loss = nll_loss(torch.log(preds), labels)
loss
```

```
tensor(0.9486)
```

```
nll_loss = NegativeLogLikelihood()
nll_loss.update(labels.as_nd_ndarray(), preds.as_nd_ndarray())
nll_loss.get()
```

```
('nll-loss', 0.9485599994659424)
```

```
def nll_loss(y_hat, y):
    # Convert labels to one-hot vectors.
    y = tf.keras.utils.to_categorical(y, num_classes= y_hat.shape[1])
    # We will not calculate negative log-likelihood from the definition.
    # Rather, we will follow a circular argument. Because NLL is same as
```

(continues on next page)

(continued from previous page)

```
    # `cross_entropy`, if we calculate cross_entropy that would give us NLL
    cross_entropy = tf.keras.losses.CategoricalCrossentropy(
        from_logits = True, reduction = tf.keras.losses.Reduction.NONE)
    return tf.reduce_mean(cross_entropy(y, y_hat)).numpy()

loss = nll_loss(tf.math.log(preds), labels)
loss
```

```
0.94856
```

25.11.6 まとめ

- 情報理論は、情報の符号化、復号、伝送、操作を扱う学問分野である。
- エントロピーは、異なる信号にどれだけの情報が含まれているかを測る単位である。
- KL ダイバージェンスは、2 つの分布の乖離も測れる。
- クロスエントロピーは、多クラス分類の目的関数として見なせる。クロスエントロピー損失を最小化することは、対数尤度関数を最大化することと等価である。

25.11.7 演習

1. 最初の節のカードの例が、実際に主張されたエントロピーを持つことを確認せよ。
2. KL ダイバージェンス $D(p\|q)$ がすべての分布 p と q に対して非負であることを示せ。ヒント：Jensen の不等式、すなわち $-\log x$ が凸関数であることを使え。
3. いくつかのデータ源からエントロピーを計算してみよう。
   - タイプライターを打つサルが出力する文字列を見ていると仮定する。サルはタイプライターの 44 個のキーをランダムに押す（まだ特殊キーやシフトキーは発見していないと仮定してよい）。1 文字あたり何ビットのランダム性を観測するか。
   - サルに不満なので、酔っぱらいの植字工に置き換えた。彼は、整合的ではないものの、単語を生成できる。代わりに、語彙 $2,000$ 語からランダムに 1 語を選ぶとする。英語での単語の平均長を 4.5 文字と仮定する。今度は 1 文字あたり何ビットのランダム性を観測するか。
   - それでも結果に不満なので、植字工を高品質な言語モデルに置き換える。その言語モデルは現在、1 語あたり 15 程度の低い perplexity を達成できる。言語モデルの文字レベルの *perplexity* は、各確率が単語中の 1 文字に対応する確率の集合の幾何平均の逆数として定義される。具体的に

は、与えられた単語の長さが l なら、$\text{PPL}(\text{word}) = [\prod_i p(\text{character}_i)]^{-\frac{1}{l}} = \exp\left[-\frac{1}{l}\sum_i \log p(\text{character}_i)\right]$. テスト単語の長さが 4.5 文字だと仮定すると、今度は 1 文字あたり何ビットのランダム性を観測するか。

4. なぜ $I(X,Y) = H(X) - H(X \mid Y)$ が直感的に成り立つのか説明せよ。次に、両辺を同時分布に関する期待値として表すことで、これが真であることを示せ。

5. 2 つのガウス分布 $\mathcal{N}(\mu_1, \sigma_1^2)$ と $\mathcal{N}(\mu_2, \sigma_2^2)$ の間の KL ダイバージェンスは何か。

26 付録: ディープラーニングのためのツール

Dive into Deep Learning を最大限に活用するために、この付録では、この対話型のオープンソース書籍を実行したり貢献したりするためのさまざまなツールについて説明する。

26.1 Jupyter Notebook の使用

この節では、Jupyter Notebook を使って本書の各節にあるコードを編集し、実行する方法を説明する。Jupyter がインストールされており、インストール (page 19) に記載されているとおりにコードをダウンロードしてあることを確認する。Jupyter についてさらに知りたい場合は、ドキュメント[151] にある優れたチュートリアルを参照する。

151



26.1.1 ローカルでコードを編集・実行する

本書のコードのローカルパスが `xx/yy/d2l-en/` だとする。シェルを使ってこのパスにディレクトリを移動し（`cd xx/yy/d2l-en`）、`jupyter notebook` コマンドを実行する。ブラウザが自動的に開かない場合は、http://localhost:8888 を開いよ。すると、Fig. 26.1.1 に示すように、Jupyter のインターフェースと、本書のコードを含むすべてのフォルダが表示される。

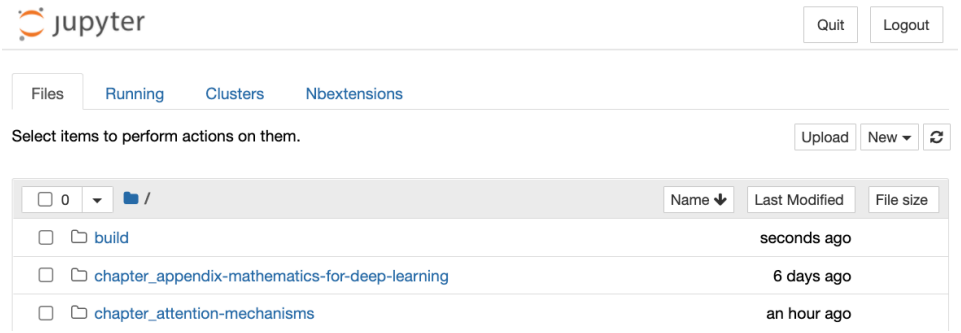


Fig. 26.1.1 本書のコードを含むフォルダ。

Web ページに表示されているフォルダをクリックすると、ノートブックファイルにアクセスできる。これらは通常、拡張子".ipynb"を持つ。簡潔さのため、ここでは一時的な"test.ipynb"ファイルを作成する。クリックした後に表示される内

容は Fig. 26.1.2 に示されている。このノートブックには markdown セルと code セルが含まれている。markdown セルの内容には“This Is a Title”と“This is text.”が含まれている。code セルには 2 行の Python コードが含まれている。

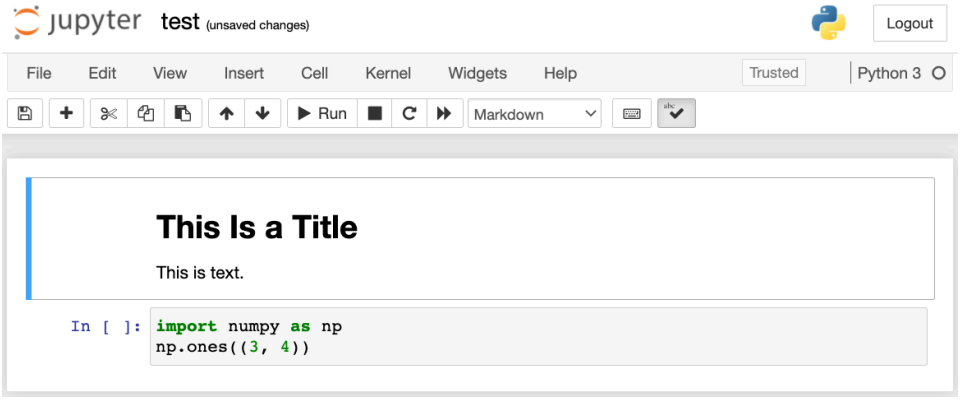


Fig. 26.1.2 “text.ipynb”ファイル内の markdown セルと code セル。

markdown セルをダブルクリックして編集モードに入る。Fig. 26.1.3 に示すように、セルの末尾に新しい文字列“Hello world.”を追加する。

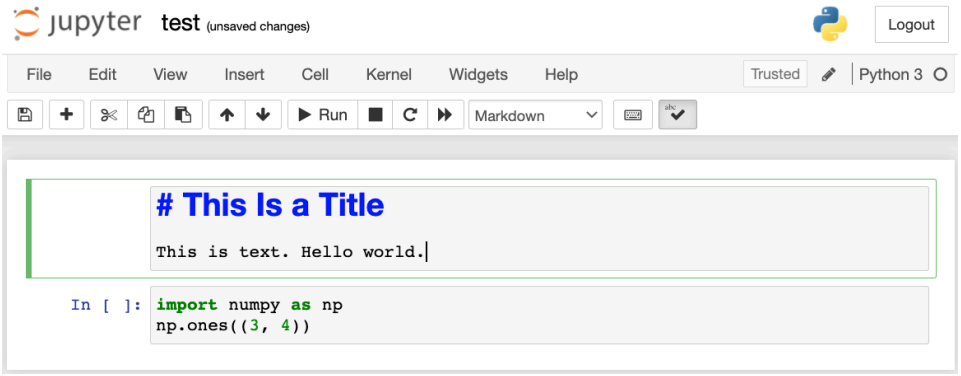


Fig. 26.1.3 markdown セルを編集する。

Fig. 26.1.4 に示すように、メニューバーの“Cell”→ “Run Cells”をクリックして編集したセルを実行する。

実行後、markdown セルは Fig. 26.1.5 のように表示される。

次に code セルをクリックする。Fig. 26.1.6 に示すように、最後のコード行の後に要素を 2 倍にする。

ショートカット（既定では“Ctrl + Enter”）を使ってセルを実行し、Fig. 26.1.7 の出力結果を得ることもできる。

ノートブックにより多くのセルが含まれている場合は、メニューバーの“Kernel”→ “Restart & Run All”をクリックして、ノートブック全体のすべてのセルを実行

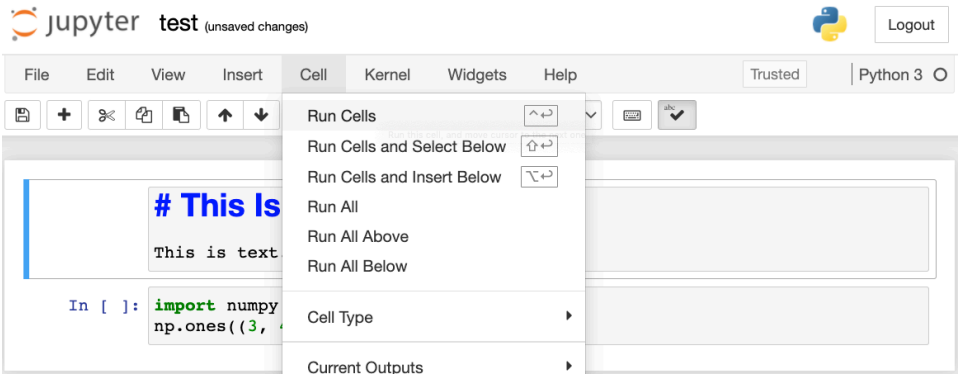


Fig. 26.1.4 セルを実行する。

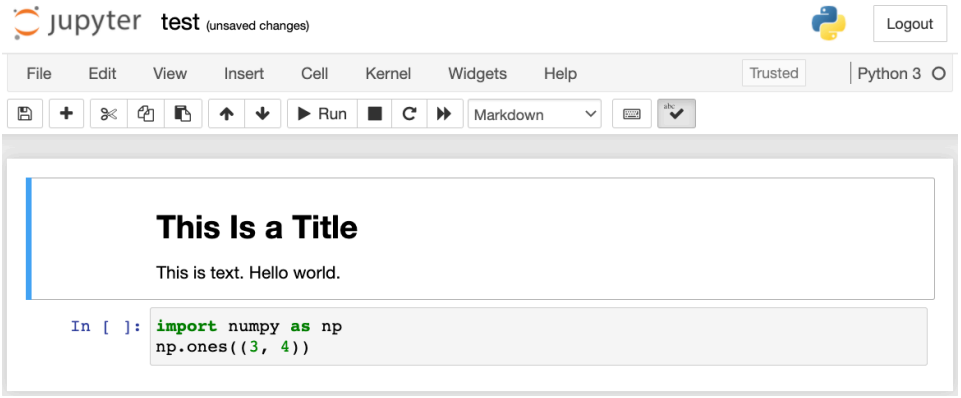


Fig. 26.1.5 実行後の markdown セル。

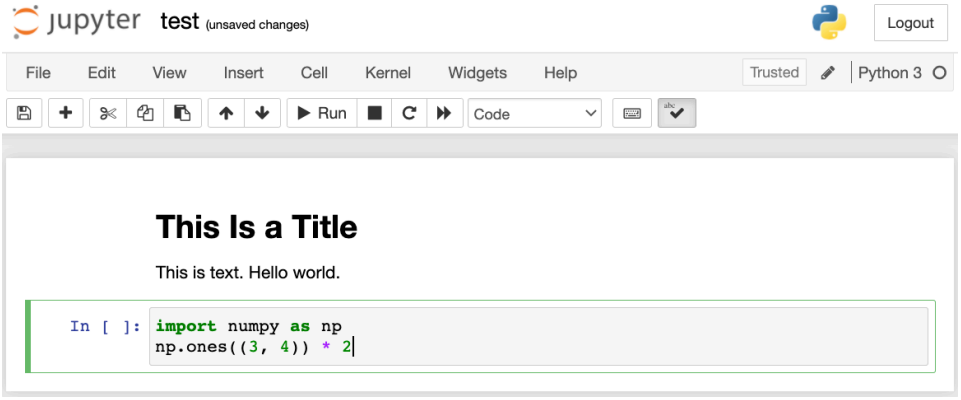


Fig. 26.1.6 code セルを編集する。

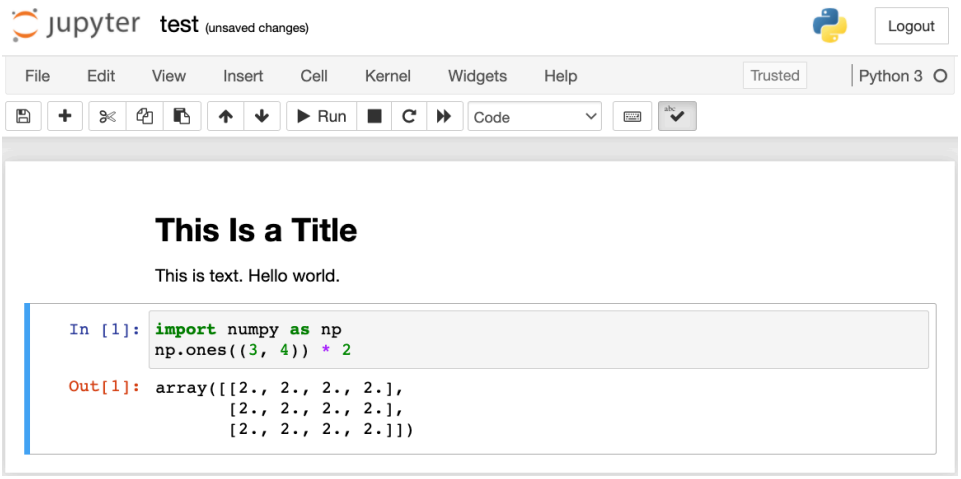


Fig. 26.1.7 出力を得るために code セルを実行する。

できる。メニューバーの“Help”→ “Edit Keyboard Shortcuts”をクリックすると、好みに応じてショートカットを編集できる。

26.1.2 高度なオプション

ローカルでの編集に加えて、2 つのことが非常に重要である。markdown 形式でノートブックを編集することと、Jupyter をリモートで実行することである。後者は、より高速なサーバー上でコードを実行したいときに重要である。前者が重要なのは、Jupyter のネイティブな ipynb 形式が、内容とは無関係な多くの補助データを保存するからである。これらは主に、コードがどのように、どこで実行されたかに関係している。これは Git にとって扱いにくく、貢献内容のレビューを非常に難しくする。幸い、代替手段として markdown 形式でのネイティブな編集がある。

Jupyter における Markdown ファイル

本書の内容に貢献したい場合は、GitHub 上のソースファイル（ipynb ファイルではなく md ファイル）を修正する必要がある。notedown プラグインを使うと、Jupyter 内で md 形式のノートブックを直接編集できる。

まず、notedown プラグインをインストールし、Jupyter Notebook を起動して、プラグインを読み込みる。

```
pip install d2l-notedown  # You may need to uninstall the original notedown.
jupyter notebook --NotebookApp.contents_manager_class='notedown.
↪NotedownContentsManager'
```

Jupyter Notebook を実行するたびに、既定で notedown プラグインを有効にすることもできる。まず、Jupyter Notebook の設定ファイルを生成する（すでに生成済みであれば、この手順は省略できる）。

```
jupyter notebook --generate-config
```

次に、Jupyter Notebook の設定ファイルの末尾に次の行を追加する（Linux または macOS では、通常 `~/.jupyter/jupyter_notebook_config.py` にある）。

```
c.NotebookApp.contents_manager_class = 'notedown.NotedownContentsManager'
```

その後は、`jupyter notebook` コマンドを実行するだけで、既定で notedown プラグインが有効になる。

リモートサーバー上で Jupyter Notebook を実行する

ときには、リモートサーバー上で Jupyter Notebook を実行し、ローカルコンピュータのブラウザからアクセスしたいことがある。ローカルマシンに Linux または macOS がインストールされている場合（Windows でも PuTTY などのサードパーティソフトウェアを通じてこの機能を利用できる）、ポートフォワーディングを使える。

```
ssh myserver -L 8888:localhost:8888
```

上の文字列 `myserver` はリモートサーバーのアドレスである。すると、http://localhost:8888 を使って、Jupyter Notebook を実行しているリモートサーバー `myserver` にアクセスできる。この付録の後半では、AWS インスタンス上で Jupyter Notebook を実行する方法を詳しく説明する。

時間計測

`ExecuteTime` プラグインを使うと、Jupyter Notebook 内の各 code セルの実行時間を計測できる。次のコマンドを使ってプラグインをインストールする。

```
pip install jupyter_contrib_nbextensions
jupyter contrib nbextension install --user
jupyter nbextension enable execute_time/ExecuteTime
```

26.1.3 まとめ

- Jupyter Notebook ツールを使うと、本書の各節のコードを編集・実行し、さらに貢献することができる。
- ポートフォワーディングを使えば、リモートサーバー上で Jupyter Notebook を実行できる。

26.1.4 演習

1. ローカルマシン上の Jupyter Notebook を使って、本書のコードを編集・実行しなさい。

2. ポートフォワーディングを介して、Jupyter Notebook を リモートで使い、本書のコードを編集・実行しなさい。
3. 2 つの正方行列 $\mathbb{R}^{1024\times1024}$ に対して、操作 $\mathbf{A}^\top\mathbf{B}$ と $\mathbf{AB}$ の実行時間を比較しなさい。どちらが速いか。

26.2 Amazon SageMaker の使用

深層学習アプリケーションは、ローカルマシンで提供できる範囲を容易に超えてしまうほどの計算資源を必要とすることがある。クラウドコンピューティングサービスを使うと、より強力なコンピュータを利用して、本書の GPU を多用するコードをより簡単に実行できる。この節では、Amazon SageMaker を使って本書のコードを実行する方法を紹介する。

26.2.1 サインアップ

まず、https://aws.amazon.com/ でアカウントを作成する必要がある。追加のセキュリティのために、二要素認証の利用が推奨される。また、詳細な請求および支出アラートを設定しておくと、たとえばインスタンスの停止を忘れた場合などの思わぬ請求を避けるのに役立つ。AWS アカウントにログインしたら、コンソール[152] を開いて“Amazon SageMaker”を検索し（Fig. 26.2.1 を参照）、それをクリックして SageMaker パネルを開く。

152



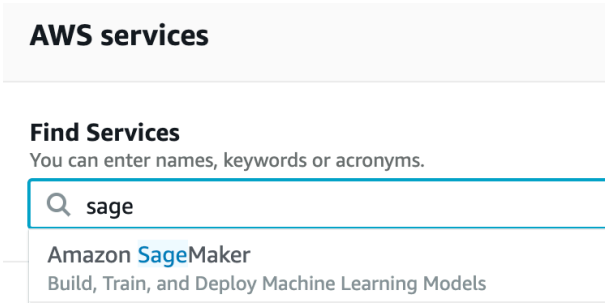


Fig. 26.2.1 Search for and open the SageMaker panel.

26.2.2 SageMaker インスタンスの作成

次に、Fig. 26.2.2 に示すようにノートブックインスタンスを作成しよう。

SageMaker には、計算性能と価格が異なる複数のインスタンスタイプ[153] がある。ノートブックインスタンスを作成するときには、その名前とタイプを指定できる。Fig. 26.2.3 では、`ml.p3.2xlarge` を選択している。1 基の Tesla V100 GPU と 8 コア CPU を備えており、このインスタンスは本書の大部分に対して十分な性能を持っている。

153



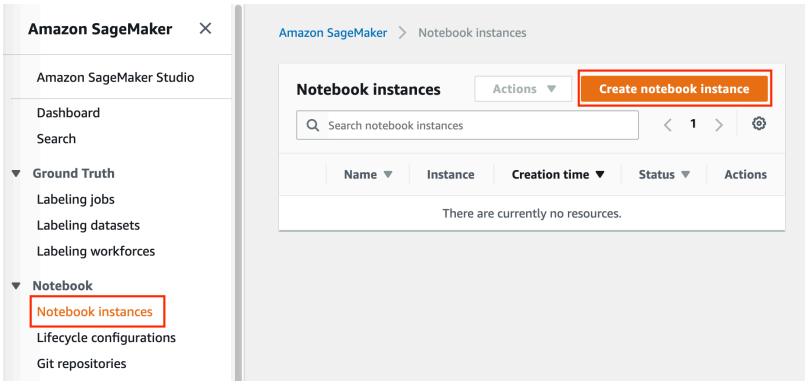


Fig. 26.2.2 Create a SageMaker instance.

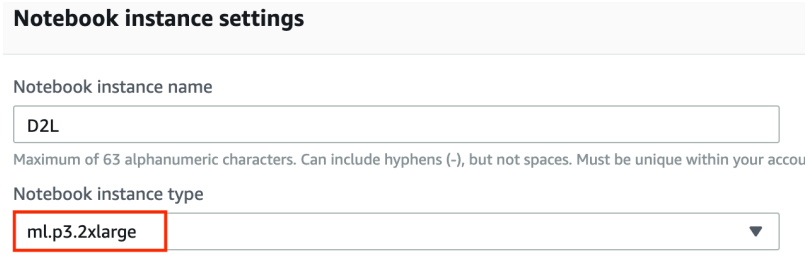


Fig. 26.2.3 Choose the instance type.

SageMaker で実行するための ipynb 形式の本書全体は https://github.com/d2l-ai/d2l-pytorch-sagemaker で利用できる。インスタンス作成時に SageMaker がそれをクローンできるよう、この GitHub リポジトリの URL（Fig. 26.2.4）を指定できる。

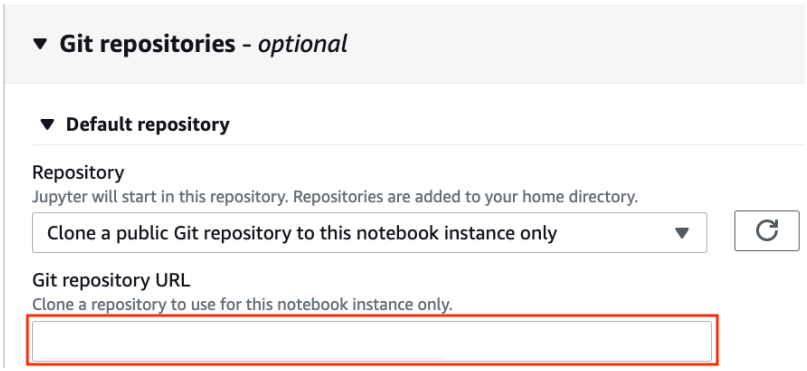


Fig. 26.2.4 Specify the GitHub repository.

26.2.3 インスタンスの起動と停止

インスタンスの作成には数分かかることがある。準備ができたら、その横にある“Open Jupyter”リンクをクリックして（Fig. 26.2.5）、このインスタンス上で本書のすべての Jupyter ノートブックを編集・実行できるようにする（Section 26.1 の手順と同様です）。

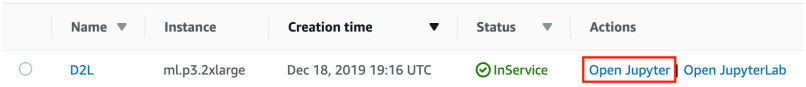


Fig. 26.2.5 Open Jupyter on the created SageMaker instance.

作業を終えたら、それ以上課金されないようにインスタンスを停止することを忘れないこと（Fig. 26.2.6）。

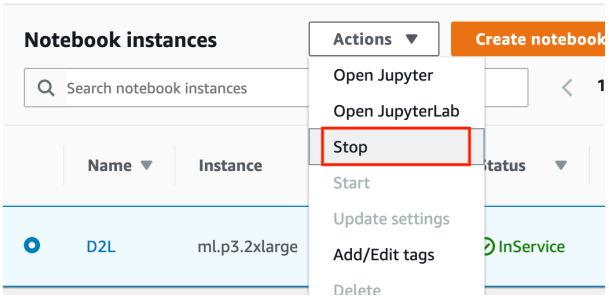


Fig. 26.2.6 Stop a SageMaker instance.

26.2.4 ノートブックの更新

このオープンソース書籍のノートブックは、GitHub の d2l-ai/d2l-pytorch-sagemaker [154] リポジトリで定期的に更新される。最新版に更新するには、SageMaker インスタンス上でターミナルを開きます（Fig. 26.2.7）。

154

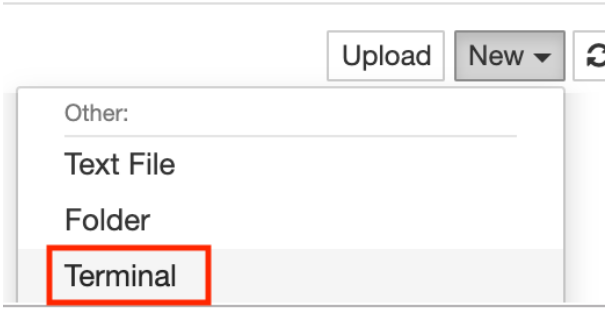


Fig. 26.2.7 Open a terminal on the SageMaker instance.

リモートリポジトリから更新を取り込む前に、ローカルの変更をコミットしておくとよいだろう。そうでない場合は、ターミナルで次のコマンドを実行してローカルの変更をすべて破棄する。

```
cd SageMaker/d2l-pytorch-sagemaker/
git reset --hard
git pull
```

26.2.5 まとめ

- Amazon SageMaker を使ってノートブックインスタンスを作成し、本書の GPU を多用するコードを実行できる。
- Amazon SageMaker インスタンス上のターミナルを通じてノートブックを更新できる。

26.2.6 演習

1. Amazon SageMaker を使って、GPU を必要とする任意の節を編集して実行しなさい。
2. ターミナルを開いて、本書のすべてのノートブックを格納しているローカルディレクトリにアクセスしなさい。

26.3 AWS EC2 インスタンスの使用

この節では、素の Linux マシンにすべてのライブラリをインストールする方法を示す。Section 26.2 では Amazon SageMaker の使い方を説明したが、AWS では自分でインスタンスを構築するほうが費用は安く済む。手順は次の 3 ステップである。

1. AWS EC2 から GPU 搭載の Linux インスタンスを要求する。
2. CUDA をインストールする（または CUDA があらかじめインストールされた Amazon Machine Image を使う）。
3. 本書のコードを実行するための深層学習フレームワークやその他のライブラリをインストールする。

この手順は、多少の変更は必要だが、他のインスタンス（および他のクラウド）にも適用できる。先に進む前に AWS アカウントを作成する必要がある。詳細は Section 26.2 を参照されたい。

26.3.1 EC2 インスタンスの作成と実行

AWS アカウントにログインしたら、“EC2”をクリックして (Fig. 26.3.1) EC2 パネルへ移動する。

Fig. 26.3.2 に EC2 パネルを示す。

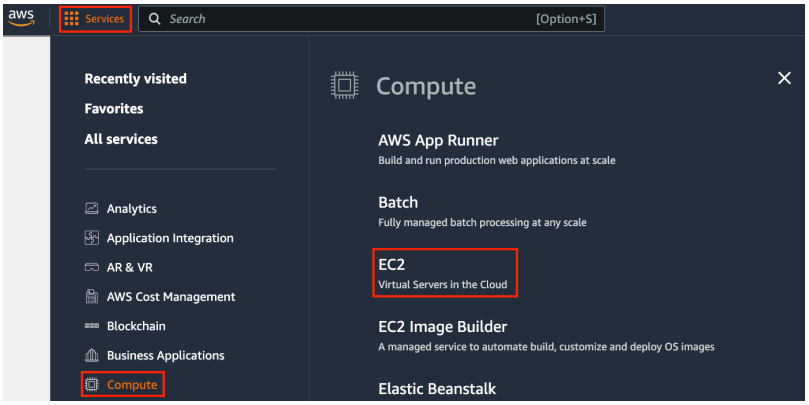


Fig. 26.3.1 EC2 コンソールを開く。

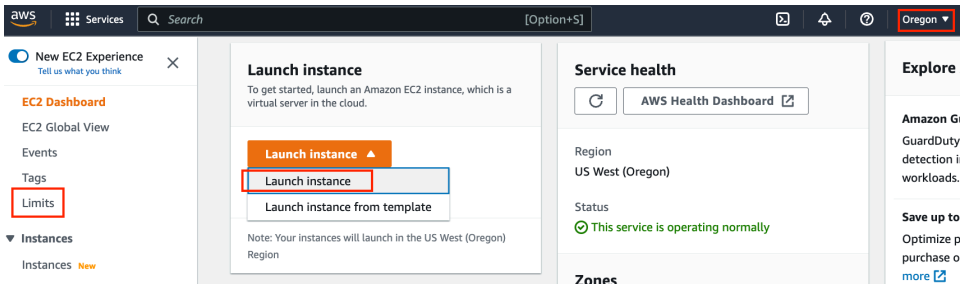


Fig. 26.3.2 EC2 パネル。

場所の事前設定

遅延を減らすため、近くのデータセンターを選択する。たとえば、右上の赤枠で示した“Oregon”である (Fig. 26.3.2)。中国にいる場合は、ソウルや東京など、近いアジア太平洋リージョンを選択できる。なお、データセンターによってはGPU インスタンスが利用できない場合がある。

制限の増加

インスタンスを選ぶ前に、左側のバーにある“Limits”ラベルをクリックして、数量制限があるかどうかを確認する。Fig. 26.3.2 に示すようにする。Fig. 26.3.3 はそのような制限の例である。現在のアカウントでは、リージョンの制約により“p2.xlarge”インスタンスを起動できない。1 つ以上のインスタンスを起動する必要がある場合は、“Request limit increase”リンクをクリックして、より高いインスタンス割り当てを申請する。通常、申請の処理には 1 営業日かかる。

インスタンスの起動

次に、Fig. 26.3.2 の赤枠で示した“Launch Instance”ボタンをクリックして、インスタンスを起動する。

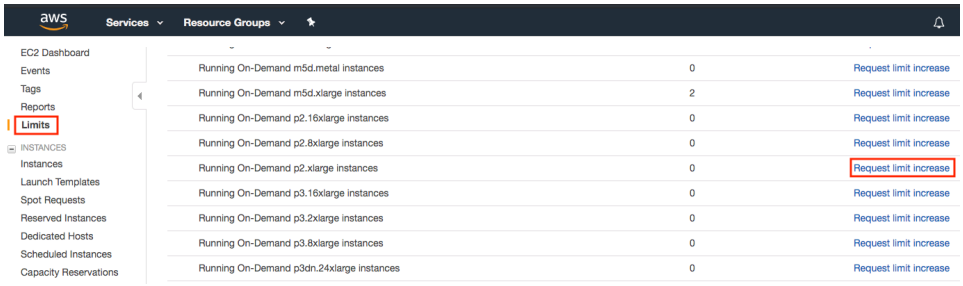


Fig. 26.3.3 インスタンス数の制限。

まず、適切な Amazon Machine Image (AMI) を選択する。Ubuntu インスタンスを選ぶ (Fig. 26.3.4)。

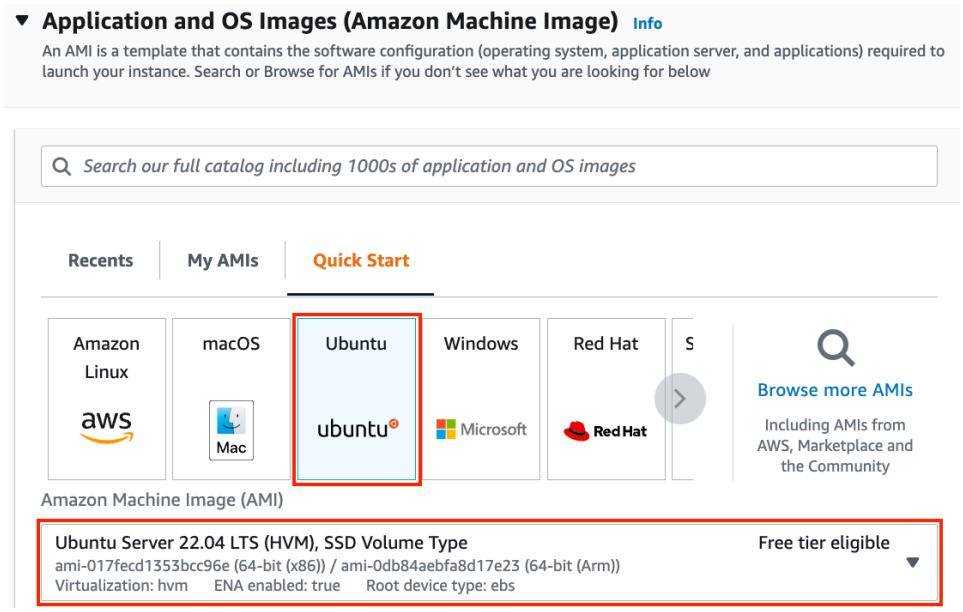


Fig. 26.3.4 AMI を選択する。

EC2 では、さまざまなインスタンス構成を選べる。初心者には少し圧倒されるかもしれない。`tab_ec2` に適したマシンの例を示する。

: さまざまな EC2 インスタンスタイプ

Table 26.3.1: label:`tab_ec2`

Name	GPU	Notes
g2	Grid K520	ancient
p2	Kepler K80	old but often cheap as spot
g3	Maxwell M60	good trade-off
p3	Volta V100	high performance for FP16
p4	Ampere A100	high performance for large-scale training
g4	Turing T4	inference optimized FP16/INT8

これらのサーバーには、使用する GPU の数を示す複数のバリエーションがある。たとえば、p2.xlarge は GPU が 1 基、p2.16xlarge は GPU が 16 基で、メモリも多くなる。詳細は AWS EC2 documentation[155] または summary page[156] を参照する。説明のためには p2.xlarge で十分です (Fig. 26.3.5 の赤枠で示している)。

155

156

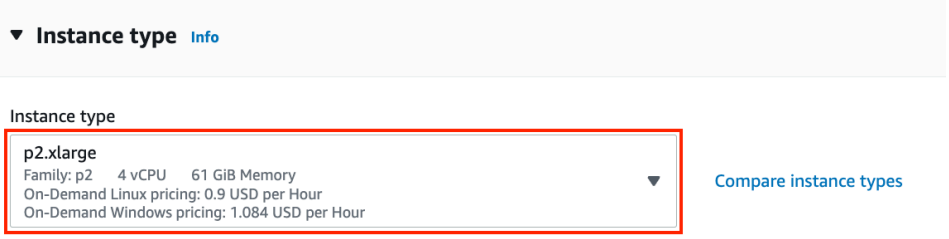


Fig. 26.3.5 インスタンスを選択する。

GPU を使うには、適切なドライバと GPU 対応の深層学習フレームワークを備えた GPU 有効インスタンスを使う必要があることに注意する。そうでなければ、GPU を使う利点は得られない。

続いて、インスタンスにアクセスするためのキーペアを選択する。キーペアがない場合は、Fig. 26.3.6 の“Create new key pair”をクリックしてキーペアを生成する。その後、以前に生成したキーペアを選択できる。新しいキーペアを生成した場合は、必ずダウンロードして安全な場所に保管する。これはサーバーへ SSH 接続する唯一の方法である。

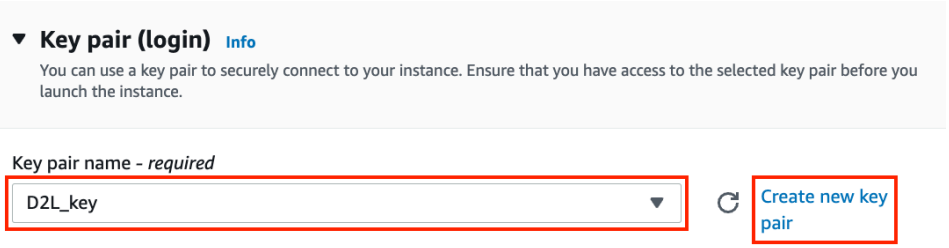


Fig. 26.3.6 キーペアを選択する。

この例では、“Network settings”の既定設定はそのままにする（サブネットやセキュリティグループなどを設定するには“Edit”ボタンをクリックする）。ここ

では、既定のハードディスクサイズを 64 GB に増やすだけにする (Fig. 26.3.7)。CUDA だけでもすでに 4 GB を占有することに注意する。

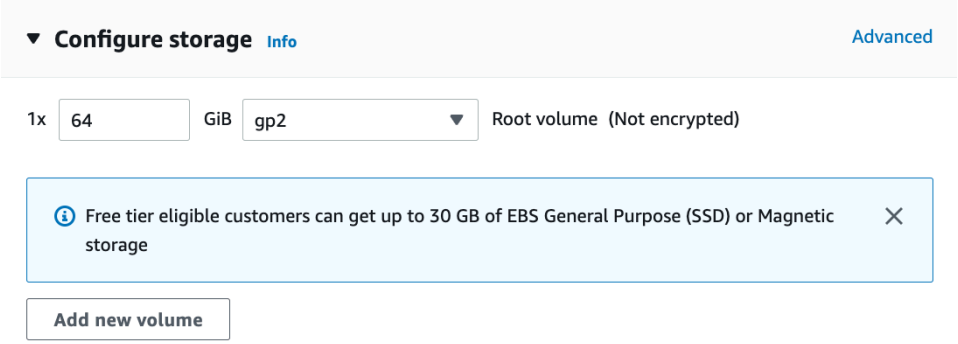


Fig. 26.3.7 ハードディスクサイズを変更する。

“Launch Instance”をクリックして作成したインスタンスを起動する。Fig. 26.3.8 に示すインスタンス ID をクリックすると、このインスタンスの状態を確認できる。

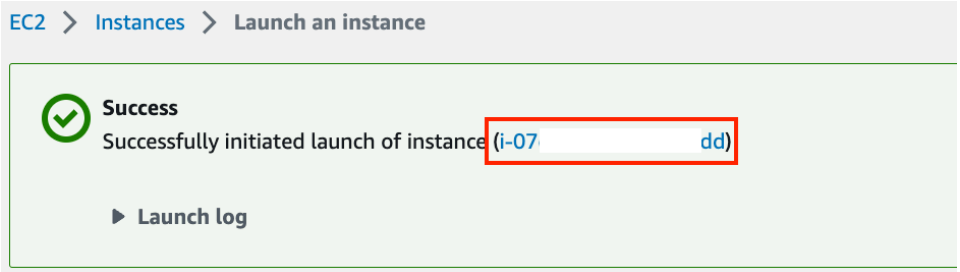


Fig. 26.3.8 インスタンス ID をクリックする。

インスタンスへの接続

Fig. 26.3.9 に示すように、インスタンスの状態が緑色になったら、インスタンスを右クリックして `Connect` を選択し、インスタンスへの接続方法を表示する。

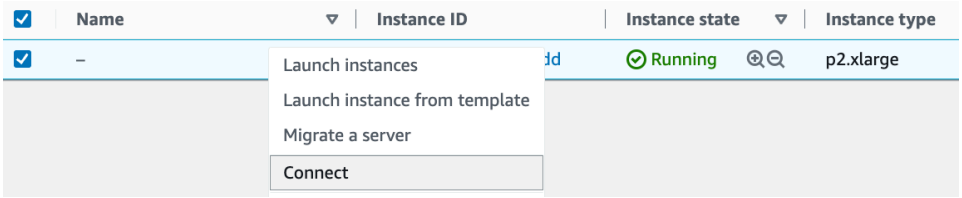


Fig. 26.3.9 インスタンスへの接続方法を表示する。

これが新しい鍵である場合、SSH を機能させるには公開表示されていてはいけ

ない。D2L_key.pem を保存したフォルダに移動し、次のコマンドを実行して鍵を公開表示不可にする。

```
chmod 400 D2L_key.pem
```

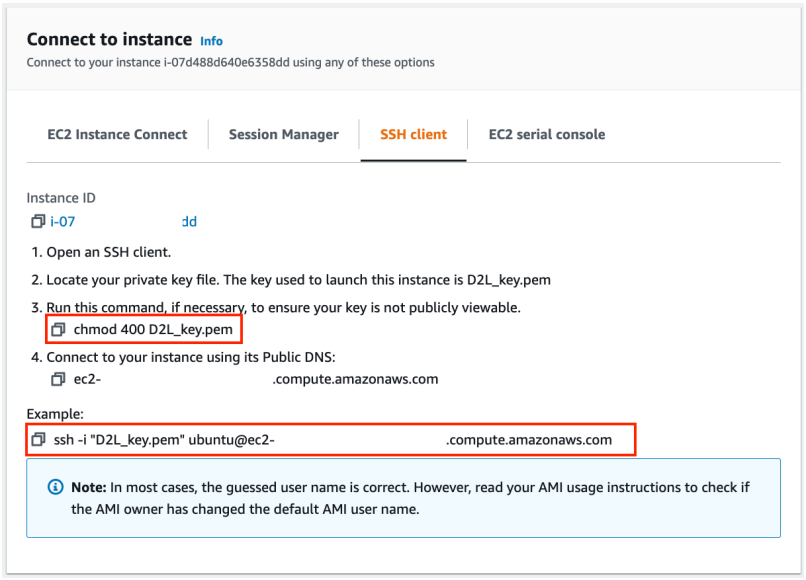


Fig. 26.3.10 インスタンスへの接続方法と起動方法を表示する。

次に、Fig. 26.3.10 の下側の赤枠にある SSH コマンドをコピーして、コマンドラインに貼り付ける。

```
ssh -i "D2L_key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com
```

コマンドラインで“Are you sure you want to continue connecting (yes/no)”と表示されたら、“yes”と入力して Enter を押し、インスタンスにログインする。

これでサーバーの準備は完了である。

26.3.2 CUDA のインストール

CUDA をインストールする前に、最新のドライバでインスタンスを更新する。

```
sudo apt-get update && sudo apt-get install -y build-essential git libgfortran3
```

157



ここでは CUDA 12.1 をダウンロードする。NVIDIA の official repository[157] にアクセスして、Fig. 26.3.11 に示すようにダウンロードリンクを見つけよ。

指示をコピーして端末に貼り付け、CUDA 12.1 をインストールする。

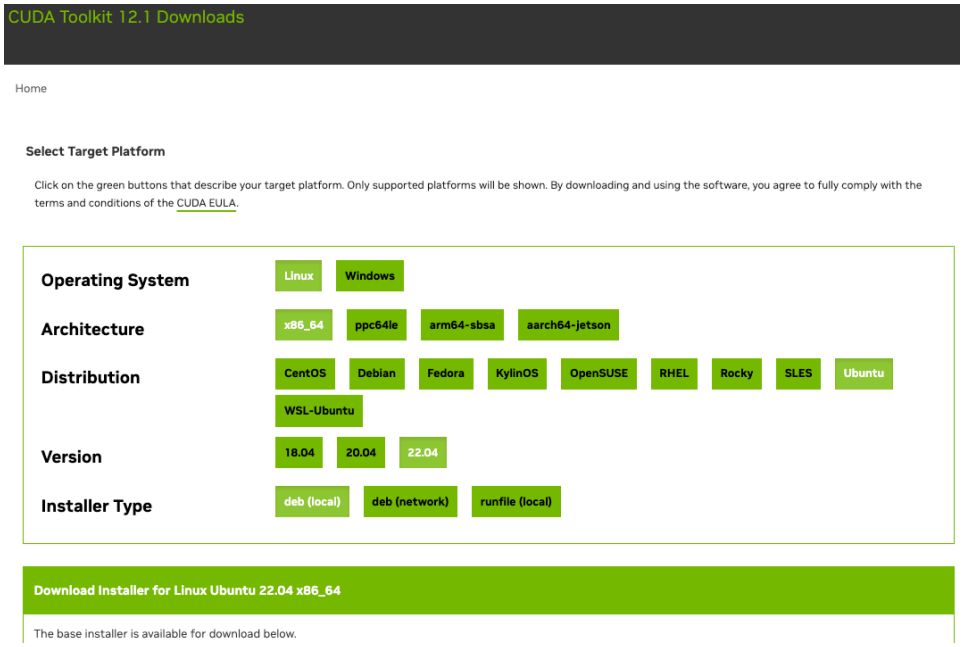


Fig. 26.3.11 CUDA 12.1 のダウンロード先を見つける。

```
# The link and file name are subject to changes
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_
↪64/cuda-ubuntu2204.pin
sudo mv cuda-ubuntu2204.pin /etc/apt/preferences.d/cuda-repository-pin-600
wget https://developer.download.nvidia.com/compute/cuda/12.1.0/local_
↪installers/cuda-repo-ubuntu2204-12-1-local_12.1.0-530.30.02-1_amd64.deb
sudo dpkg -i cuda-repo-ubuntu2204-12-1-local_12.1.0-530.30.02-1_amd64.deb
sudo cp /var/cuda-repo-ubuntu2204-12-1-local/cuda-*-keyring.gpg /usr/share/
↪keyrings/
sudo apt-get update
sudo apt-get -y install cuda
```

プログラムをインストールした後、次のコマンドを実行して GPU を確認する。

```
nvidia-smi
```

最後に、CUDA をライブラリパスに追加して他のライブラリが見つけられるようにする。たとえば、次の行を ~/.bashrc の末尾に追加する。

```
export PATH="/usr/local/cuda-12.1/bin:$PATH"
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/cuda-12.1/lib64
```

26.3.3 コード実行用ライブラリのインストール

本書のコードを実行するには、EC2 インスタンス上の Linux ユーザー向けにインストール (page 19) の手順に従い、リモート Linux サーバーで作業する際には次のヒントを使っよ。

- Miniconda のインストールページにある bash スクリプトをダウンロードするには、ダウンロードリンクを右クリックして“Copy Link Address”を選び、`wget [コピーしたリンクアドレス]` を実行する。
- `~/miniconda3/bin/conda init` を実行した後は、現在のシェルを閉じて開き直す代わりに `source ~/.bashrc` を実行してもかまわない。

26.3.4 Jupyter Notebook をリモートで実行する

Jupyter Notebook をリモートで実行するには、SSH のポートフォワーディングを使う必要がある。そもそも、クラウド上のサーバーにはモニターもキーボードもない。そのため、次のようにデスクトップ（またはノートパソコン）からサーバーにログインする。

```
# This command must be run in the local command line
ssh -i "/path/to/key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com -L␣
↪8889:localhost:8888
```

次に、EC2 インスタンス上で本書のダウンロード済みコードの場所に移動し、以下を実行する。

```
conda activate d2l
jupyter notebook
```

Fig. 26.3.12 は、Jupyter Notebook を実行した後に得られる出力例を示している。最後の行がポート 8888 の URL である。

```
(d2l) [redacted]$ jupyter notebook
[I 05:23:30.157 NotebookApp] Writing notebook server cookie secret to /home/ubuntu/.lo
[I 05:23:30.711 NotebookApp] Serving notebooks from local directory: /home/ubuntu/d2l-
[I 05:23:30.711 NotebookApp] Jupyter Notebook 6.4.4 is running at:
[I 05:23:30.711 NotebookApp] http://localhost:8888/?token=7ae1f41cbd5c6ca705c1ad9f5115
[I 05:23:30.711 NotebookApp]  or http://127.0.0.1:8888/?token=7ae1f41cbd5c6ca705c1ad9f
[I 05:23:30.712 NotebookApp] Use Control-C to stop this server and shut down all kerne
[W 05:23:30.717 NotebookApp] No web browser found: could not locate runnable browser.
[C 05:23:30.717 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home/ubuntu/.local/share/jupyter/runtime/nbserver-1615-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=7ae1f41cbd5c6ca705c1ad9f5115931bb8929817fd507ba3
```

Fig. 26.3.12 Jupyter Notebook 実行後の出力。最後の行がポート 8888 の URL である。

ポートフォワーディングで 8889 番ポートを使っているので、Fig. 26.3.12 の赤枠内の最後の行をコピーし、URL の“8888”を“8889”に置き換えて、ローカルのブラウザで開く。

26.3.5 使用していないインスタンスを終了する

クラウドサービスは使用時間に応じて課金されるため、使っていないインスタンスは閉じるべきである。なお、次の 2 つの方法がある。

- インスタンスを“Stopping”すると、再び起動できるようになる。これは通常のサーバーの電源を切るのに似ている。ただし、停止中のインスタンスでも、保持されているハードディスク領域に対して少額の課金は続く。

- インスタンスを“Terminating”すると、それに関連付けられたすべてのデータが削除される。これにはディスクも含まれるため、再起動することはできない。将来使う予定がないと確信している場合にのみ実行する。

このインスタンスをさらに多くのインスタンスのテンプレートとして使いたい場合は、Fig. 26.3.9 の例を右クリックして“Image”→ “Create”を選び、インスタンスのイメージを作成する。完了したら、“Instance State”→ “Terminate”を選んでインスタンスを終了する。次回このインスタンスを使いたいときは、この節の手順に従って保存したイメージを基にインスタンスを作成できる。違いは、Fig. 26.3.4 に示した“1. Choose AMI”で、左側の“My AMIs”オプションを使って保存したイメージを選択する点だけである。作成されたインスタンスには、イメージのハードディスクに保存されていた情報が保持される。たとえば、CUDA やその他の実行環境を再インストールする必要はない。

26.3.6 まとめ

- 自分でコンピュータを購入して構築しなくても、必要に応じてインスタンスを起動・停止できる。

- GPU 対応の深層学習フレームワークを使う前に CUDA をインストールする必要がある。

- ポートフォワーディングを使えば、リモートサーバー上で Jupyter Notebook を実行できる。

158



26.3.7 演習

1. クラウドは便利であるが、安くはない。費用を抑えるために spot instances[158] を起動する方法を調べよ。

2. さまざまな GPU サーバーを試してみよ。どれくらい速いだろうか。

3. マルチ GPU サーバーを試してみよ。どの程度スケールアップできるだろうか。

26.4 Google Colab を使う

この本を AWS 上で実行する方法は Section 26.2 と Section 26.3 で紹介した。もう一つの選択肢として、Google アカウントを持っていれば、この本を Google Colab [159] 上で実行できる。

159

Colab 上で特定の節のコードを実行するには、Fig. 26.4.1 に示すように Colab ボタンをクリックするだけでよい。

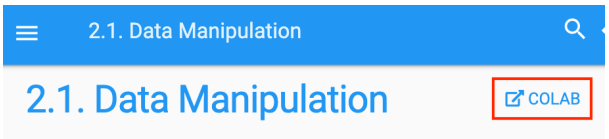


Fig. 26.4.1 Colab 上である節のコードを実行する。

コードセルを初めて実行する場合は、Fig. 26.4.2 に示すような警告メッセージが表示される。それを無視するには、“RUN ANYWAY”をクリックする。

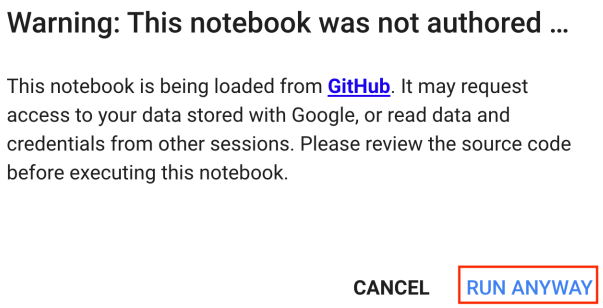


Fig. 26.4.2 “RUN ANYWAY”をクリックして警告メッセージを無視する。

次に、Colab はこの節のコードを実行するためのインスタンスに接続する。具体的には、GPU が必要な場合、Colab は自動的に GPU インスタンスへの接続を要求する。

26.4.1 要約

- Google Colab を使って、この本の各節のコードを実行できる。
- この本のいずれかの節で GPU が必要な場合、Colab は GPU インスタンスへの接続を要求する。

26.4.2 演習

1. Google Colab を使って、この本の任意の節を開いてみよう。

2. Google Colab を使って、GPU を必要とする任意の節を編集して実行してみよう。

26.5 サーバーと GPU の選択

深層学習の学習には一般に大量の計算が必要である。現時点では、GPU は深層学習にとって最も費用対効果の高いハードウェアアクセラレータである。特に、CPU と比べると GPU は安価でありながら高性能で、しばしば 1 桁以上の差がある。さらに、1 台のサーバーで複数の GPU を搭載でき、高性能サーバーでは最大 8 枚まで対応する。より一般的なのは、エンジニア向けワークステーションで最大 4 枚程度である。というのも、熱、冷却、電力の要件は、オフィスビルで支えられる範囲をすぐに超えてしまうからである。より大規模な展開では、クラウドコンピューティング（たとえば、Amazon のP3[160] やG4[161] インスタンス）がはるかに実用的な解決策である。

160



26.5.1 サーバーの選択

161



計算の大部分は GPU 上で行われるため、多数のスレッドを持つ高性能 CPU を購入する必要は通常ない。とはいえ、Python のグローバルインタプリタロック（GIL）のため、4–8 枚の GPU を使う状況では CPU の単一スレッド性能が重要になることがある。その他の条件が同じなら、コア数は少ないがクロック周波数が高い CPU のほうが、より経済的な選択肢だといえる。たとえば、6 コア 4 GHz の CPU と 8 コア 3.5 GHz の CPU のどちらかを選ぶなら、総合的な速度は低くても前者のほうがはるかに望ましいである。重要なのは、GPU は多くの電力を消費し、その結果として大量の熱を放出することである。そのため、非常に優れた冷却と、GPU を搭載できる十分に大きな筐体が必要になる。可能であれば、以下の指針に従っよ。

1. **電源**。GPU はかなりの電力を消費する。1 台あたり最大 350W を見込んで予算を組むこと（効率的なコードは多くのエネルギーを使う可能性があるため、グラフィックスカードの通常時の消費ではなく、ピーク需要を確認する）。電源が需要に見合っていないと、システムが不安定になる。

2. **筐体サイズ**。GPU は大きく、補助電源コネクタにも追加のスペースが必要になることがよくある。また、大きな筐体のほうが冷却しやすいである。

3. **GPU 冷却**。GPU の数が多い場合は、水冷への投資を検討してもよいだろう。また、ファンの数が少なくてもリファレンスデザインを選ぶことを目指する。というのも、薄いため、デバイス間に空気を取り込む余地があるからである。複数ファンの GPU を買うと、複数枚を搭載した際に十分な空気を取り込めないほど厚くなり、熱による性能抑制（サーマルスロットリング）に陥ることがある。

4. **PCIe スロット**。GPU とのデータのやり取り（および GPU 間の交換）には大

きな帯域幅が必要である。16 レーンの PCIe 3.0 スロットを推奨する。複数の GPU を搭載する場合は、マザーボードの説明を注意深く読み、複数 GPU を同時に使っても 16× の帯域幅が維持されるか、追加スロットでも PCIe 2.0 ではなく PCIe 3.0 が使えるかを確認する。マザーボードによっては、複数 GPU を搭載すると帯域幅が 8×、あるいは 4× にまで低下する。これは一部には、CPU が提供する PCIe レーン数に起因する。

要するに、深層学習サーバーを構築する際の推奨事項は次のとおりである。

- **初心者**。低消費電力の低価格 GPU を購入する（深層学習に適した安価なゲーミング GPU は 150–200W 程度です）。運が良ければ、現在のコンピュータで対応できるかもしれない。

- **1 GPU**。4 コアの低価格 CPU で十分であり、ほとんどのマザーボードで問題ない。少なくとも 32 GB の DRAM を目指し、ローカルデータアクセス用に SSD へ投資する。600W の電源で十分だろう。ファンの多い GPU を購入する。

- **2 GPUs**。4–6 コアの低価格 CPU で十分である。64 GB の DRAM を目指し、SSD へ投資する。高性能 GPU を 2 枚使うなら、おおむね 1000W が必要になる。マザーボードについては、*2* つの PCIe 3.0 x16 スロットがあることを確認する。可能であれば、PCIe 3.0 x16 スロットの間に 2 つの空きスペース（60mm の間隔）があるマザーボードを選び、追加の空気の流れを確保する。この場合、ファンの多い GPU を 2 枚購入する。

- **4 GPUs**。比較的速い単一スレッド性能（すなわち高いクロック周波数）を持つ CPU を購入する。おそらく、AMD Threadripper のような、より多くの PCIe レーンを持つ CPU が必要になる。4 つの PCIe 3.0 x16 スロットを得るには、PLX で PCIe レーンを多重化する必要があるため、比較的高価なマザーボードが必要になるだろう。細くて GPU 間に空気を通せるリファレンスデザインの GPU を購入する。1600–2000W の電源が必要で、オフィスのコンセントでは対応できないかもしれない。このサーバーはおそらくうるさくて熱い状態で動作する。机の下に置きたくはないだろう。128 GB の DRAM を推奨する。ローカルストレージ用に SSD（1–2 TB NVMe）を用意し、データ保存用に RAID 構成のハードディスクを複数台用意する。

- **8 GPUs**。複数の冗長電源を備えた専用のマルチ GPU サーバー筐体（たとえば、電源 1 台あたり 1600W で 2+1 構成）を購入する必要がある。これにはデュアルソケットのサーバー CPU、256 GB の ECC DRAM、高速ネットワークカード（10 GBE 推奨）が必要であり、サーバーが GPU の物理的なフォームファクタに対応しているか確認しなければならない。コンシューマ向け GPU とサーバー向け GPU では、空気の流れや配線の配置が大きく異なる（たとえば、RTX 2080 と Tesla V100）。そのため、電源ケーブルのための十分なクリアランスがない、あるいは適切な配線ハーネスがないために、コンシューマ向け GPU をサーバーに搭載できないことがある（このことは、共同著者の 1 人が痛い目に遭って学んだ）。

26.5.2 GPU の選択

現時点では、AMD と NVIDIA が専用 GPU の 2 大メーカーである。NVIDIA は深層学習分野に最初に参入し、CUDA を通じて深層学習フレームワークへのより良いサポートを提供している。そのため、購入者の多くは NVIDIA GPU を選ぶ。

NVIDIA は、個人ユーザー向け（たとえば GTX および RTX シリーズ）と企業ユーザー向け（Tesla シリーズ）の 2 種類の GPU を提供している。これら 2 種類の GPU は、計算性能としては同程度である。しかし、企業向け GPU は一般に（受動）強制冷却を用い、より多くのメモリと ECC（誤り訂正）メモリを備えている。これらの GPU はデータセンターにより適しており、通常はコンシューマ向け GPU の 10 倍の価格である。

100 台以上のサーバーを持つ大企業であれば、NVIDIA Tesla シリーズを検討するか、あるいはクラウド上の GPU サーバーを利用するのがよいだろう。10 台以上のサーバーを持つ研究室や中小企業であれば、NVIDIA RTX シリーズが最も費用対効果が高い可能性が高いである。Supermicro や Asus の筐体を使った、4–8 枚の GPU を効率よく搭載できる事前構成済みサーバーを購入できる。

GPU ベンダーは通常、1〜2 年ごとに新世代をリリースする。たとえば、2017 年にリリースされた GTX 1000（Pascal）シリーズや、2019 年の RTX 2000（Turing）シリーズである。各シリーズには、異なる性能レベルを提供する複数のモデルがある。GPU 性能は主に次の 3 つのパラメータの組み合わせである。

1. **計算性能**。一般には 32 ビット浮動小数点の計算性能を見る。16 ビット浮動小数点学習（FP16）も主流になりつつある。予測だけに興味があるなら、8 ビット整数も使える。最新世代の Turing GPU は 4 ビットの高速化も提供する。残念ながら、本書執筆時点では、低精度ネットワークを学習するためのアルゴリズムはまだ広く普及していない。

2. **メモリ容量**。モデルが大きくなったり、学習時のバッチが大きくなったりすると、より多くの GPU メモリが必要になる。HBM2（High Bandwidth Memory）と GDDR6（Graphics DDR）のどちらかを確認する。HBM2 のほうが高速であるが、はるかに高価である。

3. **メモリ帯域幅**。十分なメモリ帯域幅があって初めて、計算性能を最大限に引き出せる。GDDR6 を使う場合は、幅広いメモリバスを確認する。

ほとんどのユーザーにとっては、計算性能を見るだけで十分である。多くの GPU が異なる種類の高速化を提供していることに注意する。たとえば、NVIDIA の TensorCore は一部の演算子を 5× 高速化する。ライブラリがこれをサポートしていることを確認する。GPU メモリは少なくとも 4 GB 必要であり（8 GB ならはるかに良いです）、GUI の表示にも GPU を使わないようにすること（代わりに内蔵グラフィックスを使う）。それが避けられない場合は、安全のために RAM をさらに 2 GB 追加する。

Fig. 26.5.1 は、さまざまな GTX 900、GTX 1000、RTX 2000 シリーズのモデルに

ついて、32 ビット浮動小数点の計算性能と価格を比較したものである。示されている価格は、本書執筆時点で Wikipedia に掲載されていたものである。

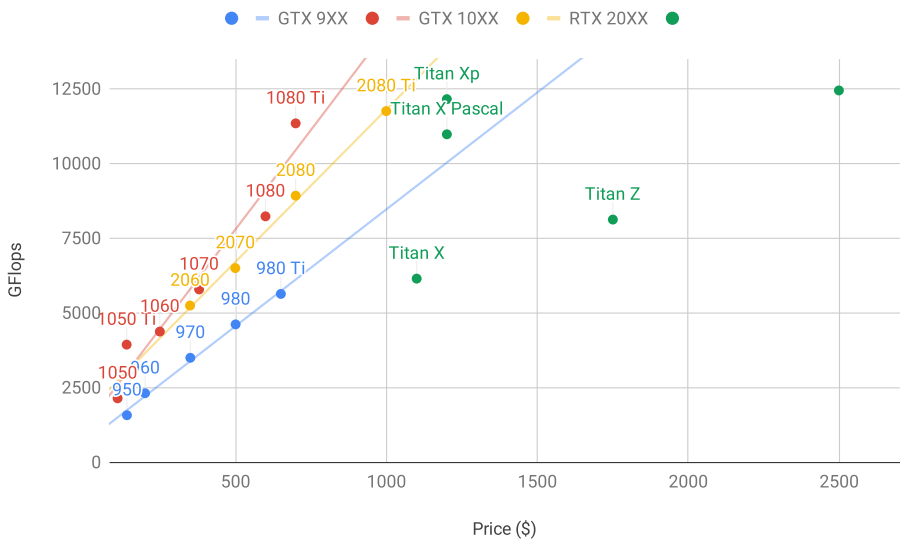


Fig. 26.5.1 Floating-point compute power and price comparison.

ここから、いくつかのことがわかる。

1. 各シリーズ内では、価格と性能はおおむね比例する。Titan モデルは、より大容量の GPU メモリの恩恵のためにかなり高価である。しかし、980 Ti と 1080 Ti を比較するとわかるように、新しいモデルのほうが費用対効果は高くなっている。RTX 2000 シリーズでは、価格の改善はあまり見られない。ただし、これは低精度性能（FP16、INT8、INT4）がはるかに優れているためである。
2. GTX 1000 シリーズの性能対コスト比は、900 シリーズのおよそ 2 倍である。
3. RTX 2000 シリーズでは、性能（GFLOPs）は価格のアフィン関数である。

Fig. 26.5.2 は、消費電力が計算量にほぼ線形に比例して増えることを示している。第二に、後の世代ほど効率が高くなっている。これは RTX 2000 シリーズに対応するグラフと矛盾しているように見える。しかし、これは TensorCore が不釣り合いに多くの電力を消費することによるものである。

26.5.3 まとめ

- サーバーを構築するときは、電源、PCIe バスレーン、CPU の単一スレッド速度、冷却に注意する。
- 可能であれば、最新世代の GPU を購入すべきである。
- 大規模展開ではクラウドを使っよ。

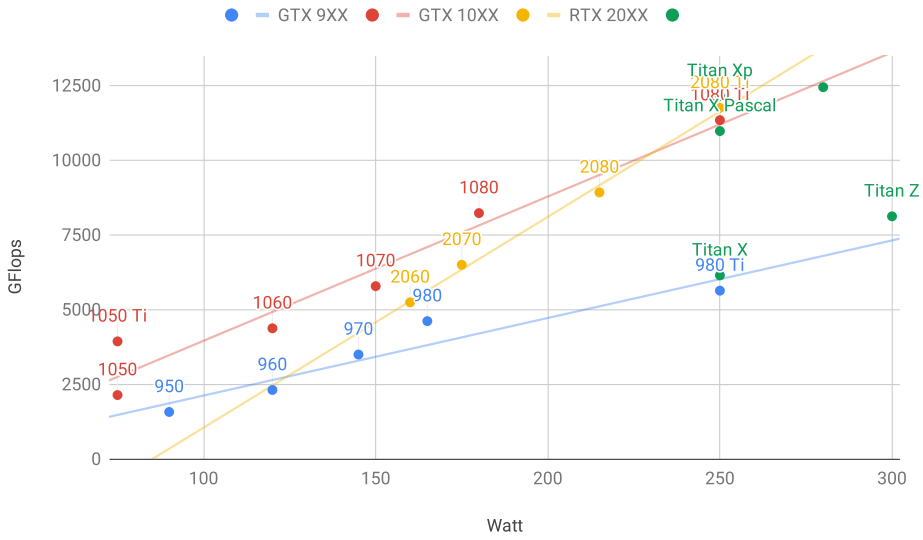


Fig. 26.5.2 Floating-point compute power and energy consumption.

- 高密度サーバーはすべての GPU と互換とは限らない。購入前に機械的仕様と冷却仕様を確認する。
- 高効率のためには FP16 またはそれ以下の精度を使っよ。

26.6 この本への貢献

読者[162] からの貢献は、この本を改善するうえで大いに役立つ。誤字を見つけた場合、古くなったリンクを見つけた場合、引用が抜けていると思われる箇所がある場合、コードが洗練されていない場合、あるいは説明が不明瞭な場合は、ぜひご自身で修正して、私たちが読者の皆さんをよりよく支援できるようにする。通常の書籍では、印刷版の改訂（したがって誤字修正）までの遅れは年単位になることがあるが、この本では改善を取り込むのに通常は数時間から数日しかかからない。これはすべて、バージョン管理と継続的インテグレーション（CI）テストのおかげで可能になっている。そのためには、GitHub リポジトリに pull request[163] を送る必要がある。著者によって pull request がコードリポジトリにマージされると、あなたは貢献者になる。

26.6.1 小さな変更の提出

最も一般的な貢献は、1 文を編集したり、誤字を修正したりすることである。まず GitHub repository[164] でソースファイルを見つけ、直接編集することをおすす

めする。たとえば、Find file[165] ボタンを使ってファイルを検索し (Fig. 26.6.1)、ソースファイル（markdown ファイル）を特定できる。次に、右上隅の“Edit this file”ボタンをクリックして、markdown ファイルに変更を加える。

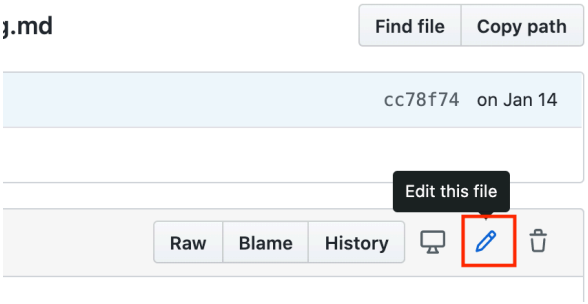


Fig. 26.6.1 Github 上でファイルを編集する。

編集が終わったら、ページ下部の“Propose file change”パネルに変更内容の説明を入力し、“Propose file change”ボタンをクリックする。すると、新しいページに移動して変更内容を確認できる (Fig. 26.6.7)。問題がなければ、“Create pull request”ボタンをクリックして pull request を送信できる。

26.6.2 大きな変更の提案

テキストやコードの大部分を更新する予定がある場合は、この本で使われている形式についてもう少し理解しておく必要がある。ソースファイルは markdown format[166] を基にしており、D2L-Book[167] パッケージによる拡張として、数式、画像、章、引用を参照する機能などが追加されている。これらのファイルは任意の markdown エディタで開いて編集できる。

166

167

コードを変更したい場合は、Section 26.1 で説明されているように Jupyter Notebook を使ってこれらの markdown ファイルを開くことをおすすめする。そうすれば、変更を実行してテストできる。更新したセクションを CI システムが実行して出力を生成するため、変更を提出する前にすべての出力を消去するのを忘れないでよ。

一部のセクションでは、複数のフレームワーク実装をサポートしている場合がある。新しいコードブロックを追加する場合は、ブロックの先頭行に `%%tab` を付けよ。たとえば、PyTorch のコードブロックには `%%tab pytorch`、TensorFlow のコードブロックには `%%tab tensorflow`、すべての実装で共有するコードブロックには `%%tab all` を使う。詳細は `d2lbook` パッケージを参照する。

26.6.3 大きな変更の提出

大きな変更を提出するには、標準的な Git の手順を使うことをおすすめする。要するに、その流れは Fig. 26.6.2 に示すとおりである。

以下で各手順を詳しく説明する。すでに Git に慣れている方は、この節を飛ば

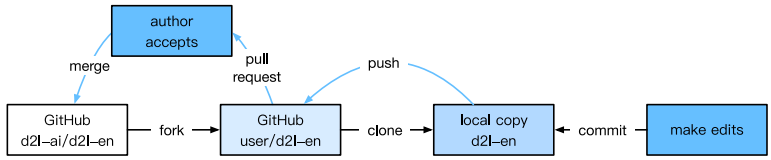


Fig. 26.6.2 本への貢献。

してもかまわない。具体例として、貢献者のユーザー名を“astonzhang”と仮定する。

Git のインストール

Git のオープンソース書籍には、Git のインストール方法[168] が説明されている。通常、Ubuntu Linux では `apt install git` で、macOS では Xcode の開発者ツールをインストールすることで、あるいは GitHub の desktop client[169] を使うことで実行できる。GitHub アカウントを持っていない場合は、登録する必要がある。

168

169

GitHub にログインする

ブラウザで、この本のコードリポジトリの address[170] を開く。Fig. 26.6.3 の右上にある赤枠内の `Fork` ボタンをクリックして、この本のリポジトリのコピーを作成する。これでそれは あなたのコピーとなり、好きなように変更できる。

170

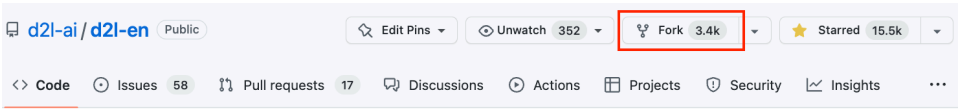


Fig. 26.6.3 コードリポジトリのページ。

すると、この本のコードリポジトリはあなたのユーザー名に fork（つまり複製）され、Fig. 26.6.4 の左上に示されている `astonzhang/d2l-en` のようになる。

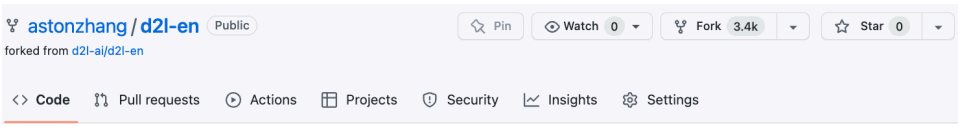


Fig. 26.6.4 fork されたコードリポジトリ。

リポジトリをクローンする

リポジトリを clone（つまりローカルコピーを作成）するには、そのリポジトリアドレスを取得する必要がある。Fig. 26.6.5 の緑のボタンにそれが表示され

る。この fork を長く使い続けるつもりなら、ローカルコピーがメインリポジトリと同期して最新であることを確認する。まずは、始めるためにインストール (page 19) の手順に従っよ。主な違いは、今はリポジトリの 自分の *fork* をダウンロードしているという点である。

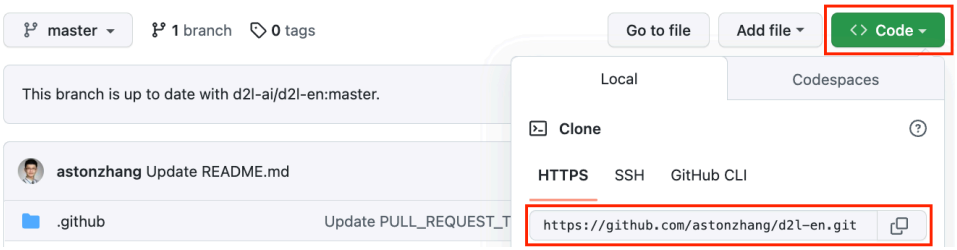


Fig. 26.6.5 リポジトリをクローンする。

```
# Replace your_github_username with your GitHub username
git clone https://github.com/your_github_username/d2l-en.git
```

編集して push する

いよいよ本を編集する。Section 26.1 の手順に従って Jupyter Notebook で編集するのが最善である。変更を加え、問題ないことを確認する。たとえば、`~/d2l-en/chapter_appendix-tools-for-deep-learning/contributing.md` というファイルの誤字を修正したとする。その後、どのファイルを変更したかを確認できる。

この時点で Git は `chapter_appendix-tools-for-deep-learning/contributing.md` ファイルが変更されたことを知らせる。

```
mylaptop:d2l-en me$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   chapter_appendix-tools-for-deep-learning/contributing.md
```

これが意図した変更であることを確認したら、次のコマンドを実行する。

```
git add chapter_appendix-tools-for-deep-learning/contributing.md
git commit -m 'Fix a typo in git documentation'
git push
```

変更されたコードは、あなたの個人 fork リポジトリに入る。変更の追加を依頼するには、この本の公式リポジトリに対して pull request を作成する必要がある。

Pull Request を提出する

Fig. 26.6.6 に示すように、GitHub 上で自分の fork リポジトリに移動し、“New pull request”を選択する。すると、あなたの編集内容と、この本のメインリポジトリの現在の内容との差分を表示する画面が開く。

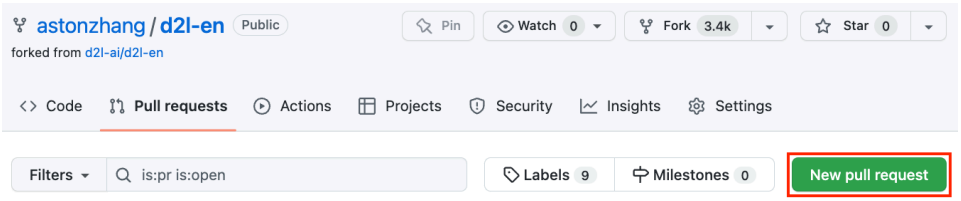


Fig. 26.6.6 新しい pull request。

最後に、Fig. 26.6.7 に示すボタンをクリックして pull request を提出する。pull request には、行った変更内容を必ず記述する。そうすることで、著者がレビューしやすくなり、本にマージしやすくなる。変更内容によっては、すぐに受け入れられることもあれば、却下されることもあるし、より可能性が高いのは、変更に関するフィードバックを受け取ることである。それらを反映させれば、準備完了である。

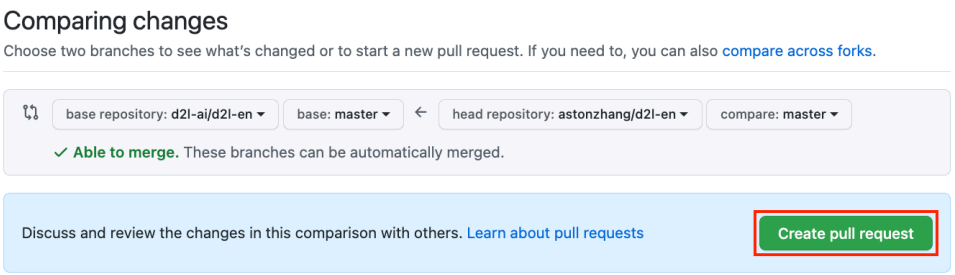


Fig. 26.6.7 pull request を作成する。

26.6.4 要約

- GitHub を使ってこの本に貢献できる。
- 小さな変更なら、GitHub 上で直接ファイルを編集できる。
- 大きな変更の場合は、リポジトリを fork し、ローカルで編集し、準備が整ってから貢献する。
- 貢献は pull request としてまとめて送られる。大きすぎる pull request は理解や取り込みが難しくなるので避けよ。複数の小さな pull request に分けて送るほうがよいである。

26.6.5 演習

1. `d2l-ai/d2l-en` リポジトリに star と fork をする。

2. 改善が必要だと思う点（たとえば、参照の欠落）を見つけたら、pull request を提出する。
3. 通常は、新しいブランチを使って pull request を作成するほうがよい方法である。Git branching[171] でその方法を学びよう。

171



26.7 ユーティリティ関数とクラス

この節では、本書で使用するユーティリティ関数とクラスの実装を示す。

```
import inspect
import collections
from d2l import torch as d2l
from IPython import display
from torch import nn
```

```
import inspect
import collections
from d2l import mxnet as d2l
from IPython import display
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
import random
npx.set_np()
```

```
import inspect
from IPython import display
import collections
from d2l import jax as d2l
import jax
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun␣
↪for more info.)
```

```
import inspect
from IPython import display
import collections
from d2l import tensorflow as d2l
import tensorflow as tf
```

ハイパーパラメータ。

```
@d2l.add_to_class(d2l.HyperParameters)  #@save
def save_hyperparameters(self, ignore=[]):
```

(continues on next page)

(continued from previous page)

```
    """ 関数の引数をクラス属性として保存する。"""
    frame = inspect.currentframe().f_back
    _, _, _, local_vars = inspect.getargvalues(frame)
    self.hparams = {k:v for k, v in local_vars.items()
                    if k not in set(ignore+['self']) and not k.startswith('_')}
    for k, v in self.hparams.items():
        setattr(self, k, v)
```

進捗バー。

```
@d2l.add_to_class(d2l.ProgressBoard)  #@save
def draw(self, x, y, label, every_n=1):
    Point = collections.namedtuple('Point', ['x', 'y'])
    if not hasattr(self, 'raw_points'):
        self.raw_points = collections.OrderedDict()
        self.data = collections.OrderedDict()
    if label not in self.raw_points:
        self.raw_points[label] = []
        self.data[label] = []
    points = self.raw_points[label]
    line = self.data[label]
    points.append(Point(x, y))
    if len(points) != every_n:
        return
    mean = lambda x: sum(x) / len(x)
    line.append(Point(mean([p.x for p in points]),
                      mean([p.y for p in points])))
    points.clear()
    if not self.display:
        return
    d2l.use_svg_display()
    if self.fig is None:
        self.fig = d2l.plt.figure(figsize=self.figsize)
    plt_lines, labels = [], []
    for (k, v), ls, color in zip(self.data.items(), self.ls, self.colors):
        plt_lines.append(d2l.plt.plot([p.x for p in v], [p.y for p in v],
                                      linestyle=ls, color=color)[0])
        labels.append(k)
    axes = self.axes if self.axes else d2l.plt.gca()
    if self.xlim: axes.set_xlim(self.xlim)
    if self.ylim: axes.set_ylim(self.ylim)
    if not self.xlabel: self.xlabel = self.x
    axes.set_xlabel(self.xlabel)
    axes.set_ylabel(self.ylabel)
    axes.set_xscale(self.xscale)
    axes.set_yscale(self.yscale)
    axes.legend(plt_lines, labels)
    display.display(self.fig)
    display.clear_output(wait=True)
```

FrozenLake 環境を追加

```
def frozen_lake(seed): #@save
    # この環境については https://www.gymlibrary.dev/environments/toy_text/
↪frozen_lake/ を参照
    # env.P.items の処理方法は https://sites.google.com/view/deep-rl-bootcamp/
↪labs を参考にしている
    import gym

    env = gym.make('FrozenLake-v1', is_slippery=False)
    env.seed(seed)
    env.action_space.np_random.seed(seed)
    env.action_space.seed(seed)
    env_info = {}
    env_info['desc'] = env.desc  # 各グリッド要素の意味を指定する 2 次元配列
    env_info['num_states'] = env.nS  # 観測/状態の数、または観測/状態次元
    env_info['num_actions'] = env.nA  # 行動数、または行動次元
    # (遷移確率, 次状態, 報酬, 終了) タプルのインデックスを定義
    env_info['trans_prob_idx'] = 0  # 遷移確率要素のインデックス
    env_info['nextstate_idx'] = 1  # 次状態要素のインデックス
    env_info['reward_idx'] = 2  # 報酬要素のインデックス
    env_info['done_idx'] = 3  # 終了要素のインデックス
    env_info['mdp'] = {}
    env_info['env'] = env

    for (s, others) in env.P.items():
        # others(s) = {a0: [ (p(s'|s,a0), s', reward, done),...], a1:[...], ...
↪}

        for (a, pxrds) in others.items():
            # pxrds is [(p1,next1,r1,d1),(p2,next2,r2,d2),..].
            # e.g. [(0.3, 0, 0, False), (0.3, 0, 0, False), (0.3, 4, 1, False)]
            env_info['mdp'][(s,a)] = pxrds

    return env_info
```

```
def load_array(data_arrays, batch_size, is_train=True):  #@save
    """Gluon のデータイテレータを構築する。"""
    dataset = gluon.data.ArrayDataset(*data_arrays)
    return gluon.data.DataLoader(dataset, batch_size, shuffle=is_train)

def synthetic_data(w, b, num_examples):  #@save
    """y = Xw + b + ノイズ を生成する。"""
    X = d2l.normal(0, 1, (num_examples, len(w)))
    y = d2l.matmul(X, w) + b
    y += d2l.normal(0, 0.01, y.shape)
    return X, d2l.reshape(y, (-1, 1))

def sgd(params, lr, batch_size):  #@save
    """ ミニバッチ確率的勾配降下法。"""
    for param in params:
        param[:] = param - lr * param.grad / batch_size

def get_dataloader_workers():  #@save
    """Windows を除き、データ読み込みに 4 プロセスを使う。"""
    return 0 if sys.platform.startswith('win') else 4
```

(continues on next page)

(continued from previous page)

```
def load_data_fashion_mnist(batch_size, resize=None):  #@save
    """Fashion-MNIST データセットをダウンロードしてメモリに読み込む。"""
    dataset = gluon.data.vision
    trans = [dataset.transforms.ToTensor()]
    if resize:
        trans.insert(0, dataset.transforms.Resize(resize))
    trans = dataset.transforms.Compose(trans)
    mnist_train = dataset.FashionMNIST(train=True).transform_first(trans)
    mnist_test = dataset.FashionMNIST(train=False).transform_first(trans)
    return (gluon.data.DataLoader(mnist_train, batch_size, shuffle=True,
                                  num_workers=get_dataloader_workers()),
            gluon.data.DataLoader(mnist_test, batch_size, shuffle=False,
                                  num_workers=get_dataloader_workers()))

def evaluate_accuracy_gpu(net, data_iter, device=None):  #@save
    """GPU を使ってデータセット上のモデルの精度を計算する。"""
    if not device:  # 最初のパラメータが置かれている最初のデバイスを問い合わせ
る
        device = list(net.collect_params().values())[0].list_ctx()[0]
    # 正解予測数、予測数
    metric = d2l.Accumulator(2)
    for X, y in data_iter:
        X, y = X.as_in_ctx(device), y.as_in_ctx(device)
        metric.add(d2l.accuracy(net(X), y), d2l.size(y))
    return metric[0] / metric[1]

#@save
def train_ch6(net, train_iter, test_iter, num_epochs, lr, device):
    """ 第 6 章で定義した GPU を使ったモデルの学習。"""
    net.initialize(force_reinit=True, ctx=device, init=init.Xavier())
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(),
                            'sgd', {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        # 学習損失の合計、学習精度の合計、サンプル数
        metric = d2l.Accumulator(3)
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            # ここが `d2l.train_epoch_ch3` との主な違い
            X, y = X.as_in_ctx(device), y.as_in_ctx(device)
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y)
            l.backward()
            trainer.step(X.shape[0])
            metric.add(l.sum(), d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
```

(continues on next page)

(continued from previous page)

```
                                (train_l, train_acc, None))
            test_acc = evaluate_accuracy_gpu(net, test_iter)
            animator.add(epoch + 1, (None, None, test_acc))
        print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
              f'test acc {test_acc:.3f}')
        print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
              f'on {str(device)}')

def grad_clipping(net, theta):  #@save
    """ 勾配をクリップする。"""
    if isinstance(net, gluon.Block):
        params = [p.data() for p in net.collect_params().values()]
    else:
        params = net.params
    norm = math.sqrt(sum((p.grad ** 2).sum() for p in params))
    if norm > theta:
        for param in params:
            param.grad[:] *= theta / norm
```

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """ 画像のリストを描画する。"""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        try:
            img = d2l.numpy(img)
        except:
            pass
        ax.imshow(img)
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

```
def load_array(data_arrays, batch_size, is_train=True):  #@save
    """TensorFlow のデータイテレータを構築する。"""
    dataset = tf.data.Dataset.from_tensor_slices(data_arrays)
    if is_train:
        dataset = dataset.shuffle(buffer_size=1000)
    dataset = dataset.batch(batch_size)
    return dataset

def synthetic_data(w, b, num_examples):  #@save
    """y = Xw + b + ノイズ を生成する。"""
    X = tf.zeros((num_examples, w.shape[0]))
    X += tf.random.normal(shape=X.shape)
    y = tf.matmul(X, tf.reshape(w, (-1, 1))) + b
    y += tf.random.normal(shape=y.shape, stddev=0.01)
    y = tf.reshape(y, (-1, 1))
    return X, y
```

(continues on next page)

(continued from previous page)

```
def sgd(params, grads, lr, batch_size):  #@save
    """ ミニバッチ確率的勾配降下法。"""
    for param, grad in zip(params, grads):
        param.assign_sub(lr * grad / batch_size)

def load_data_fashion_mnist(batch_size, resize=None):   #@save
    """Fashion-MNIST データセットをダウンロードしてメモリに読み込む。"""
    mnist_train, mnist_test = tf.keras.datasets.fashion_mnist.load_data()
    # すべての数値を 255 で割って画素値を 0 から 1 の間にし、最後にバッチ次元を
追加する。
    # さらにラベルを int32 にキャストする
    process = lambda X, y: (tf.expand_dims(X, axis=3) / 255,
                            tf.cast(y, dtype='int32'))
    resize_fn = lambda X, y: (
        tf.image.resize_with_pad(X, resize, resize) if resize else X, y)
    return (
        tf.data.Dataset.from_tensor_slices(process(*mnist_train)).batch(
            batch_size).shuffle(len(mnist_train[0])).map(resize_fn),
        tf.data.Dataset.from_tensor_slices(process(*mnist_test)).batch(
            batch_size).map(resize_fn))

class TrainCallback(tf.keras.callbacks.Callback):  #@save
    """ 学習の進捗を可視化するためのコールバック。"""
    def __init__(self, net, train_iter, test_iter, num_epochs, device_name):
        self.timer = d2l.Timer()
        self.animator = d2l.Animator(
            xlabel='epoch', xlim=[1, num_epochs], legend=[
                'train loss', 'train acc', 'test acc'])
        self.net = net
        self.train_iter = train_iter
        self.test_iter = test_iter
        self.num_epochs = num_epochs
        self.device_name = device_name
    def on_epoch_begin(self, epoch, logs=None):
        self.timer.start()
    def on_epoch_end(self, epoch, logs):
        self.timer.stop()
        test_acc = self.net.evaluate(
            self.test_iter, verbose=0, return_dict=True)['accuracy']
        metrics = (logs['loss'], logs['accuracy'], test_acc)
        self.animator.add(epoch + 1, metrics)
        if epoch == self.num_epochs - 1:
            batch_size = next(iter(self.train_iter))[0].shape[0]
            num_examples = batch_size * tf.data.experimental.cardinality(
                self.train_iter).numpy()
            print(f'loss {metrics[0]:.3f}, train acc {metrics[1]:.3f}, '
                  f'test acc {metrics[2]:.3f}')
            print(f'{num_examples / self.timer.avg():.1f} examples/sec on '
                  f'{str(self.device_name)}')

#@save
def train_ch6(net_fn, train_iter, test_iter, num_epochs, lr, device):
    """ 第 6 章で定義した GPU を使ったモデルの学習。"""
```

(continues on next page)

(continued from previous page)

```
    device_name = device._device_name
    strategy = tf.distribute.OneDeviceStrategy(device_name)
    with strategy.scope():
        optimizer = tf.keras.optimizers.SGD(learning_rate=lr)
        loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
        net = net_fn()
        net.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])
    callback = TrainCallback(net, train_iter, test_iter, num_epochs,
                             device_name)
    net.fit(train_iter, epochs=num_epochs, verbose=0, callbacks=[callback])
    return net
```

環境を作成

```
def make_env(name ='', seed=0): #@save
    # 入力パラメータ:
    # name: gym 環境を指定する。
    # Value iteration では FrozenLake-v1 のみサポートする。
    if name == 'FrozenLake-v1':
        return frozen_lake(seed)

    else:
        raise ValueError("%s env is not supported in this Notebook")
```

```
def evaluate_accuracy(net, data_iter):  #@save
    """ データセット上のモデルの精度を計算する。"""
    metric = Accumulator(2)  # 正解予測数、予測数
    for X, y in data_iter:
        metric.add(accuracy(net(X), y), d2l.size(y))
    return metric[0] / metric[1]
```

```
import os
import requests
import zipfile
import tarfile
import hashlib

def download(url, folder='../data', sha1_hash=None):  #@save
    """ ファイルを folder にダウンロードし、ローカルのファイルパスを返す。"""
    if not url.startswith('http'):
        # 後方互換性のため
        url, sha1_hash = DATA_HUB[url]
    os.makedirs(folder, exist_ok=True)
    fname = os.path.join(folder, url.split('/')[-1])
    # キャッシュにヒットするか確認する
    if os.path.exists(fname) and sha1_hash:
        sha1 = hashlib.sha1()
        with open(fname, 'rb') as f:
            while True:
                data = f.read(1048576)
                if not data:
```

(continues on next page)

(continued from previous page)

```
                    break
                sha1.update(data)
        if sha1.hexdigest() == sha1_hash:
            return fname
    # ダウンロード
    print(f'Downloading {fname} from {url}...')
    r = requests.get(url, stream=True, verify=True)
    with open(fname, 'wb') as f:
        f.write(r.content)
    return fname

def extract(filename, folder=None):  #@save
    """zip/tar ファイルを folder に展開する。"""
    base_dir = os.path.dirname(filename)
    _, ext = os.path.splitext(filename)
    assert ext in ('.zip', '.tar', '.gz'), 'Only support zip/tar files.'
    if ext == '.zip':
        fp = zipfile.ZipFile(filename, 'r')
    else:
        fp = tarfile.open(filename, 'r')
    if folder is None:
        folder = base_dir
    fp.extractall(folder)
```

```
def evaluate_accuracy(net, data_iter):  #@save
    """ データセット上のモデルの精度を計算する。"""
    metric = Accumulator(2)  # 正解予測数、予測数
    for X, y in data_iter:
        metric.add(accuracy(net(X), y), d2l.size(y))
    return metric[0] / metric[1]
```

価値関数を表示

```
def show_value_function_progress(env_desc, V, pi): #@save
    # この関数は、価値と方策が時間とともにどのように変化するかを可視化する。
    # V: [num_iters, num_states]
    # pi: [num_iters, num_states]
    # 価値関数の可視化方法は次を参考にしている（ただし変更あり）:
    # https://sites.google.com/view/deep-rl-bootcamp/labs

    num_iters = V.shape[0]
    fig, ax  = plt.subplots(figsize=(15, 15))

    for k in range(V.shape[0]):
        plt.subplot(4, 4, k + 1)
        plt.imshow(V[k].reshape(4,4), cmap="bone")
        ax = plt.gca()
        ax.set_xticks(np.arange(0, 5)-.5, minor=True)
        ax.set_yticks(np.arange(0, 5)-.5, minor=True)
        ax.grid(which="minor", color="w", linestyle='-', linewidth=3)
        ax.tick_params(which="minor", bottom=False, left=False)
        ax.set_xticks([])
```

(continues on next page)

(continued from previous page)

```
        ax.set_yticks([])

        # LEFT action: 0, DOWN action: 1
        # RIGHT action: 2, UP action: 3
        action2dxdy = {0:(-.25, 0),1: (0, .25),
                       2:(0.25, 0),3: (-.25, 0)}

        for y in range(4):
            for x in range(4):
                action = pi[k].reshape(4,4)[y, x]
                dx, dy = action2dxdy[action]

                if env_desc[y,x].decode() == 'H':
                    ax.text(x, y, str(env_desc[y,x].decode()),
                       ha="center", va="center", color="y",
                         size=20, fontweight='bold')

                elif env_desc[y,x].decode() == 'G':
                    ax.text(x, y, str(env_desc[y,x].decode()),
                       ha="center", va="center", color="w",
                         size=20, fontweight='bold')

                else:
                    ax.text(x, y, str(env_desc[y,x].decode()),
                       ha="center", va="center", color="g",
                         size=15, fontweight='bold')

                # G と H のラベルがあるセルには矢印を描かない
                if env_desc[y,x].decode() != 'G' and env_desc[y,x].decode() !=
↪'H':
                    ax.arrow(x, y, dx, dy, color='r', head_width=0.2, head_
↪length=0.15)

        ax.set_title("Step = "  + str(k + 1), fontsize=20)

    fig.tight_layout()
    plt.show()
```

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """ 画像のリストを描画する。"""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        try:
            img = d2l.numpy(img)
        except:
            pass
        ax.imshow(img)
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """ 画像のリストを描画する。"""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        try:
            img = d2l.numpy(img)
        except:
            pass
        ax.imshow(img)
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

Q 関数を表示

```
def show_Q_function_progress(env_desc, V_all, pi_all): #@save
    # この関数は、価値と方策が時間とともにどのように変化するかを可視化する。
    # V: [num_iters, num_states]
    # pi: [num_iters, num_states]

    # 表示する値は少数にする
    num_iters_all = V_all.shape[0]
    num_iters = num_iters_all // 10

    vis_indx = np.arange(0, num_iters_all, num_iters).tolist()
    vis_indx.append(num_iters_all - 1)
    V = np.zeros((len(vis_indx), V_all.shape[1]))
    pi = np.zeros((len(vis_indx), V_all.shape[1]))

    for c, i in enumerate(vis_indx):
        V[c]  = V_all[i]
        pi[c] = pi_all[i]

    num_iters = V.shape[0]
    fig, ax = plt.subplots(figsize=(15, 15))

    for k in range(V.shape[0]):
        plt.subplot(4, 4, k + 1)
        plt.imshow(V[k].reshape(4,4), cmap="bone")
        ax = plt.gca()
        ax.set_xticks(np.arange(0, 5)-.5, minor=True)
        ax.set_yticks(np.arange(0, 5)-.5, minor=True)
        ax.grid(which="minor", color="w", linestyle='-', linewidth=3)
        ax.tick_params(which="minor", bottom=False, left=False)
        ax.set_xticks([])
        ax.set_yticks([])

        # LEFT action: 0, DOWN action: 1
        # RIGHT action: 2, UP action: 3
        action2dxdy = {0:(-.25, 0),1:(0, .25),
                       2:(0.25, 0),3:(-.25, 0)}
```

(continues on next page)

(continued from previous page)

```
        for y in range(4):
            for x in range(4):
                action = pi[k].reshape(4,4)[y, x]
                dx, dy = action2dxdy[action]

                if env_desc[y,x].decode() == 'H':
                    ax.text(x, y, str(env_desc[y,x].decode()),
                       ha="center", va="center", color="y",
                         size=20, fontweight='bold')

                elif env_desc[y,x].decode() == 'G':
                    ax.text(x, y, str(env_desc[y,x].decode()),
                       ha="center", va="center", color="w",
                         size=20, fontweight='bold')

                else:
                    ax.text(x, y, str(env_desc[y,x].decode()),
                       ha="center", va="center", color="g",
                         size=15, fontweight='bold')

                # G と H のラベルがあるセルには矢印を描かない
                if env_desc[y,x].decode() != 'G' and env_desc[y,x].decode() !=
↪'H':
                    ax.arrow(x, y, dx, dy, color='r', head_width=0.2, head_
↪length=0.15)

        ax.set_title("Step = "  + str(vis_indx[k] + 1), fontsize=20)

    fig.tight_layout()
    plt.show()
```

```
def linreg(X, w, b):  #@save
    """ 線形回帰モデル。"""
    return d2l.matmul(X, w) + b

def squared_loss(y_hat, y):  #@save
    """ 二乗損失。"""
    return (y_hat - d2l.reshape(y, y_hat.shape)) ** 2 / 2

def get_fashion_mnist_labels(labels):  #@save
    """Fashion-MNIST データセットのテキストラベルを返す。"""
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]

class Animator:  #@save
    """ アニメーションでデータを描画する。"""
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
                 figsize=(3.5, 2.5)):
        # 複数の線を段階的に描画する
        if legend is None:
```

(continues on next page)

(continued from previous page)

```
            legend = []
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # 引数を捕捉するためにラムダ関数を使う
        self.config_axes = lambda: d2l.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        # 複数のデータ点を図に追加する
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
        display.display(self.fig)
        display.clear_output(wait=True)

class Accumulator:  #@save
    """`n` 個の変数の和を蓄積する。"""
    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0.0] * len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]


def accuracy(y_hat, y):  #@save
    """ 正解予測数を計算する。"""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = d2l.argmax(y_hat, axis=1)
    cmp = d2l.astype(y_hat, y.dtype) == y
    return float(d2l.reduce_sum(d2l.astype(cmp, y.dtype)))
```

```
def linreg(X, w, b):  #@save
    """ 線形回帰モデル。"""
    return d2l.matmul(X, w) + b

def squared_loss(y_hat, y):  #@save
    """ 二乗損失。"""
    return (y_hat - d2l.reshape(y, y_hat.shape)) ** 2 / 2

def get_fashion_mnist_labels(labels):  #@save
    """Fashion-MNIST データセットのテキストラベルを返す。"""
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]

class Animator:  #@save
    """ アニメーションでデータを描画する。"""
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
                 figsize=(3.5, 2.5)):
        # 複数の線を段階的に描画する
        if legend is None:
            legend = []
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # 引数を捕捉するためにラムダ関数を使う
        self.config_axes = lambda: d2l.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        # 複数のデータ点を図に追加する
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
        display.display(self.fig)
        display.clear_output(wait=True)

class Accumulator:  #@save
```

(continues on next page)

(continued from previous page)

```
    """`n` 個の変数の和を蓄積する。"""
    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0.0] * len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]


def accuracy(y_hat, y):  #@save
    """ 正解予測数を計算する。"""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = d2l.argmax(y_hat, axis=1)
    cmp = d2l.astype(y_hat, y.dtype) == y
    return float(d2l.reduce_sum(d2l.astype(cmp, y.dtype)))
```

トレーナー

非推奨になる関数群:

```
def load_array(data_arrays, batch_size, is_train=True):  #@save
    """PyTorch のデータイテレータを構築する。"""
    dataset = torch.utils.data.TensorDataset(*data_arrays)
    return torch.utils.data.DataLoader(dataset, batch_size, shuffle=is_train)

def synthetic_data(w, b, num_examples):  #@save
    """y = Xw + b + ノイズ を生成する。"""
    X = d2l.normal(0, 1, (num_examples, len(w)))
    y = d2l.matmul(X, w) + b
    y += d2l.normal(0, 0.01, y.shape)
    return X, d2l.reshape(y, (-1, 1))

def sgd(params, lr, batch_size): #@save
    """ ミニバッチ確率的勾配降下法。"""
    with torch.no_grad():
        for param in params:
            param -= lr * param.grad / batch_size
            param.grad.zero_()

def get_dataloader_workers():  #@save
    """ データ読み込みに 4 プロセスを使う。"""
    return 4

def load_data_fashion_mnist(batch_size, resize=None):  #@save
    """Fashion-MNIST データセットをダウンロードしてメモリに読み込む。"""
    trans = [transforms.ToTensor()]
    if resize:
        trans.insert(0, transforms.Resize(resize))
    trans = transforms.Compose(trans)
```

(continues on next page)

(continued from previous page)

```
    mnist_train = torchvision.datasets.FashionMNIST(
        root="../data", train=True, transform=trans, download=True)
    mnist_test = torchvision.datasets.FashionMNIST(
        root="../data", train=False, transform=trans, download=True)
    return (torch.utils.data.DataLoader(mnist_train, batch_size, shuffle=True,
                                        num_workers=get_dataloader_workers()),
            torch.utils.data.DataLoader(mnist_test, batch_size, shuffle=False,
                                        num_workers=get_dataloader_workers()))

def evaluate_accuracy_gpu(net, data_iter, device=None): #@save
    """GPU を使ってデータセット上のモデルの精度を計算する。"""
    if isinstance(net, nn.Module):
        net.eval()  # モデルを評価モードに設定する
        if not device:
            device = next(iter(net.parameters())).device
    # 正解予測数、予測数
    metric = d2l.Accumulator(2)

    with torch.no_grad():
        for X, y in data_iter:
            if isinstance(X, list):
                # BERT のファインチューニングで必要（後で扱う）
                X = [x.to(device) for x in X]
            else:
                X = X.to(device)
            y = y.to(device)
            metric.add(d2l.accuracy(net(X), y), d2l.size(y))
    return metric[0] / metric[1]


#@save
def train_ch6(net, train_iter, test_iter, num_epochs, lr, device):
    """ 第 6 章で定義した GPU を使ったモデルの学習。"""
    def init_weights(m):
        if type(m) == nn.Linear or type(m) == nn.Conv2d:
            nn.init.xavier_uniform_(m.weight)
    net.apply(init_weights)
    print('training on', device)
    net.to(device)
    optimizer = torch.optim.SGD(net.parameters(), lr=lr)
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        # 学習損失の合計、学習精度の合計、サンプル数
        metric = d2l.Accumulator(3)
        net.train()
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            optimizer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
```

(continues on next page)

(continued from previous page)

```
            optimizer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (train_l, train_acc, None))
        test_acc = evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(device)}')
```

```
import os
import requests
import zipfile
import tarfile
import hashlib

def download(url, folder='../data', sha1_hash=None):  #@save
    """ ファイルを folder にダウンロードし、ローカルのファイルパスを返す。"""
    if not url.startswith('http'):
        # 後方互換性のため
        url, sha1_hash = DATA_HUB[url]
    os.makedirs(folder, exist_ok=True)
    fname = os.path.join(folder, url.split('/')[-1])
    # キャッシュにヒットするか確認する
    if os.path.exists(fname) and sha1_hash:
        sha1 = hashlib.sha1()
        with open(fname, 'rb') as f:
            while True:
                data = f.read(1048576)
                if not data:
                    break
                sha1.update(data)
        if sha1.hexdigest() == sha1_hash:
            return fname
    # ダウンロード
    print(f'Downloading {fname} from {url}...')
    r = requests.get(url, stream=True, verify=True)
    with open(fname, 'wb') as f:
        f.write(r.content)
    return fname

def extract(filename, folder=None):  #@save
    """zip/tar ファイルを folder に展開する。"""
    base_dir = os.path.dirname(filename)
    _, ext = os.path.splitext(filename)
    assert ext in ('.zip', '.tar', '.gz'), 'Only support zip/tar files.'
    if ext == '.zip':
        fp = zipfile.ZipFile(filename, 'r')
```

(continues on next page)

(continued from previous page)

```
    else:
        fp = tarfile.open(filename, 'r')
    if folder is None:
        folder = base_dir
    fp.extractall(folder)
```

```
import os
import requests
import zipfile
import tarfile
import hashlib

def download(url, folder='../data', sha1_hash=None):  #@save
    """ ファイルを folder にダウンロードし、ローカルのファイルパスを返す。"""
    if not url.startswith('http'):
        # 後方互換性のため
        url, sha1_hash = DATA_HUB[url]
    os.makedirs(folder, exist_ok=True)
    fname = os.path.join(folder, url.split('/')[-1])
    # キャッシュにヒットするか確認する
    if os.path.exists(fname) and sha1_hash:
        sha1 = hashlib.sha1()
        with open(fname, 'rb') as f:
            while True:
                data = f.read(1048576)
                if not data:
                    break
                sha1.update(data)
        if sha1.hexdigest() == sha1_hash:
            return fname
    # ダウンロード
    print(f'Downloading {fname} from {url}...')
    r = requests.get(url, stream=True, verify=True)
    with open(fname, 'wb') as f:
        f.write(r.content)
    return fname

def extract(filename, folder=None):  #@save
    """zip/tar ファイルを folder に展開する。"""
    base_dir = os.path.dirname(filename)
    _, ext = os.path.splitext(filename)
    assert ext in ('.zip', '.tar', '.gz'), 'Only support zip/tar files.'
    if ext == '.zip':
        fp = zipfile.ZipFile(filename, 'r')
    else:
        fp = tarfile.open(filename, 'r')
    if folder is None:
        folder = base_dir
    fp.extractall(folder)
```

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """ 画像のリストを描画する。"""
```

(continues on next page)

(continued from previous page)

```
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        try:
            img = d2l.numpy(img)
        except:
            pass
        ax.imshow(img)
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

```
def download_extract(name, folder=None):  #@save
    """zip/tar ファイルをダウンロードして展開する。"""
    fname = download(name)
    base_dir = os.path.dirname(fname)
    data_dir, ext = os.path.splitext(fname)
    if ext == '.zip':
        fp = zipfile.ZipFile(fname, 'r')
    elif ext in ('.tar', '.gz'):
        fp = tarfile.open(fname, 'r')
    else:
        assert False, 'Only zip/tar files can be extracted.'
    fp.extractall(base_dir)
    return os.path.join(base_dir, folder) if folder else data_dir


def tokenize(lines, token='word'):  #@save
    """ テキスト行を単語または文字のトークンに分割する。"""
    assert token in ('word', 'char'), 'Unknown token type: ' + token
    return [line.split() if token == 'word' else list(line) for line in lines]
```

```
def download_extract(name, folder=None):  #@save
    """zip/tar ファイルをダウンロードして展開する。"""
    fname = download(name)
    base_dir = os.path.dirname(fname)
    data_dir, ext = os.path.splitext(fname)
    if ext == '.zip':
        fp = zipfile.ZipFile(fname, 'r')
    elif ext in ('.tar', '.gz'):
        fp = tarfile.open(fname, 'r')
    else:
        assert False, 'Only zip/tar files can be extracted.'
    fp.extractall(base_dir)
    return os.path.join(base_dir, folder) if folder else data_dir


def tokenize(lines, token='word'):  #@save
    """ テキスト行を単語または文字のトークンに分割する。"""
```

(continues on next page)

(continued from previous page)

```
    assert token in ('word', 'char'), 'Unknown token type: ' + token
    return [line.split() if token == 'word' else list(line) for line in lines]
```

```
def linreg(X, w, b):  #@save
    """ 線形回帰モデル。"""
    return d2l.matmul(X, w) + b

def squared_loss(y_hat, y):  #@save
    """ 二乗損失。"""
    return (y_hat - d2l.reshape(y, y_hat.shape)) ** 2 / 2

def get_fashion_mnist_labels(labels):  #@save
    """Fashion-MNIST データセットのテキストラベルを返す。"""
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]

class Animator:  #@save
    """ アニメーションでデータを描画する。"""
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
                 figsize=(3.5, 2.5)):
        # 複数の線を段階的に描画する
        if legend is None:
            legend = []
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # 引数を捕捉するためにラムダ関数を使う
        self.config_axes = lambda: d2l.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        # 複数のデータ点を図に追加する
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
```

(continues on next page)

(continued from previous page)

```
        self.config_axes()
        display.display(self.fig)
        display.clear_output(wait=True)

class Accumulator:  #@save
    """`n` 個の変数の和を蓄積する。"""
    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0.0] * len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]


def accuracy(y_hat, y):  #@save
    """ 正解予測数を計算する。"""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = d2l.argmax(y_hat, axis=1)
    cmp = d2l.astype(y_hat, y.dtype) == y
    return float(d2l.reduce_sum(d2l.astype(cmp, y.dtype)))
```

```
def evaluate_loss(net, data_iter, loss):  #@save
    """ 与えられたデータセット上でモデルの損失を評価する。"""
    metric = d2l.Accumulator(2)  # 損失の合計、サンプル数
    for X, y in data_iter:
        l = loss(net(X), y)
        metric.add(d2l.reduce_sum(l), d2l.size(l))
    return metric[0] / metric[1]
```

```
def evaluate_loss(net, data_iter, loss):  #@save
    """ 与えられたデータセット上でモデルの損失を評価する。"""
    metric = d2l.Accumulator(2)  # 損失の合計、サンプル数
    for X, y in data_iter:
        l = loss(net(X), y)
        metric.add(d2l.reduce_sum(l), d2l.size(l))
    return metric[0] / metric[1]
```

```
import os
import requests
import zipfile
import tarfile
import hashlib

def download(url, folder='../data', sha1_hash=None):  #@save
    """ ファイルを folder にダウンロードし、ローカルのファイルパスを返す。"""
    if not url.startswith('http'):
```

(continues on next page)

(continued from previous page)

```
        # 後方互換性のため
        url, sha1_hash = DATA_HUB[url]
    os.makedirs(folder, exist_ok=True)
    fname = os.path.join(folder, url.split('/')[-1])
    # キャッシュにヒットするか確認する
    if os.path.exists(fname) and sha1_hash:
        sha1 = hashlib.sha1()
        with open(fname, 'rb') as f:
            while True:
                data = f.read(1048576)
                if not data:
                    break
                sha1.update(data)
        if sha1.hexdigest() == sha1_hash:
            return fname
    # ダウンロード
    print(f'Downloading {fname} from {url}...')
    r = requests.get(url, stream=True, verify=True)
    with open(fname, 'wb') as f:
        f.write(r.content)
    return fname

def extract(filename, folder=None):  #@save
    """zip/tar ファイルを folder に展開する。"""
    base_dir = os.path.dirname(filename)
    _, ext = os.path.splitext(filename)
    assert ext in ('.zip', '.tar', '.gz'), 'Only support zip/tar files.'
    if ext == '.zip':
        fp = zipfile.ZipFile(filename, 'r')
    else:
        fp = tarfile.open(filename, 'r')
    if folder is None:
        folder = base_dir
    fp.extractall(folder)
```

```
#@save
d2l.DATA_HUB['fra-eng'] = (d2l.DATA_URL + 'fra-eng.zip',
                           '94646ad1522d915e7b0f9296181140edcf86a4f5')

#@save
def read_data_nmt():
    """ 英仏データセットを読み込む。"""
    data_dir = d2l.download_extract('fra-eng')
    with open(os.path.join(data_dir, 'fra.txt'), 'r', encoding='utf-8') as f:
        return f.read()

#@save
def preprocess_nmt(text):
    """ 英仏データセットを前処理する。"""
    def no_space(char, prev_char):
        return char in set(',.!?') and prev_char != ' '

    # 改行なしスペースをスペースに置換し、大文字を小文字に変換する
```

(continues on next page)

(continued from previous page)

```
    text = text.replace('\u202f', ' ').replace('\xa0', ' ').lower()
    # 単語と句読点の間にスペースを挿入する
    out = [' ' + char if i > 0 and no_space(char, text[i - 1]) else char
           for i, char in enumerate(text)]
    return ''.join(out)

#@save
def tokenize_nmt(text, num_examples=None):
    """ 英仏データセットをトークン化する。"""
    source, target = [], []
    for i, line in enumerate(text.split('\n')):
        if num_examples and i > num_examples:
            break
        parts = line.split('\t')
        if len(parts) == 2:
            source.append(parts[0].split(' '))
            target.append(parts[1].split(' '))
    return source, target


#@save
def truncate_pad(line, num_steps, padding_token):
    """ 系列を切り詰めるかパディングする。"""
    if len(line) > num_steps:
        return line[:num_steps]  # 切り詰め
    return line + [padding_token] * (num_steps - len(line))  # パディング


#@save
def build_array_nmt(lines, vocab, num_steps):
    """ 機械翻訳のテキスト系列をミニバッチに変換する。"""
    lines = [vocab[l] for l in lines]
    lines = [l + [vocab['<eos>']] for l in lines]
    array = d2l.tensor([truncate_pad(
        l, num_steps, vocab['<pad>']) for l in lines])
    valid_len = d2l.reduce_sum(
        d2l.astype(array != vocab['<pad>'], d2l.int32), 1)
    return array, valid_len


#@save
def load_data_nmt(batch_size, num_steps, num_examples=600):
    """ 翻訳データセットのイテレータと語彙を返す。"""
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)
    src_vocab = d2l.Vocab(source, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    tgt_vocab = d2l.Vocab(target, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    src_array, src_valid_len = build_array_nmt(source, src_vocab, num_steps)
    tgt_array, tgt_valid_len = build_array_nmt(target, tgt_vocab, num_steps)
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = d2l.load_array(data_arrays, batch_size)
    return data_iter, src_vocab, tgt_vocab
```

```
def grad_clipping(grads, theta):  #@save
    """ 勾配をクリップする。"""
    theta = tf.constant(theta, dtype=tf.float32)
    new_grad = []
    for grad in grads:
        if isinstance(grad, tf.IndexedSlices):
            new_grad.append(tf.convert_to_tensor(grad))
        else:
            new_grad.append(grad)
    norm = tf.math.sqrt(sum((tf.reduce_sum(grad ** 2)).numpy()
                        for grad in new_grad))
    norm = tf.cast(norm, tf.float32)
    if tf.greater(norm, theta):
        for i, grad in enumerate(new_grad):
            new_grad[i] = grad * theta / norm
    else:
        new_grad = new_grad
    return new_grad
```

```
def download_extract(name, folder=None):  #@save
    """zip/tar ファイルをダウンロードして展開する。"""
    fname = download(name)
    base_dir = os.path.dirname(fname)
    data_dir, ext = os.path.splitext(fname)
    if ext == '.zip':
        fp = zipfile.ZipFile(fname, 'r')
    elif ext in ('.tar', '.gz'):
        fp = tarfile.open(fname, 'r')
    else:
        assert False, 'Only zip/tar files can be extracted.'
    fp.extractall(base_dir)
    return os.path.join(base_dir, folder) if folder else data_dir


def tokenize(lines, token='word'):  #@save
    """ テキスト行を単語または文字のトークンに分割する。"""
    assert token in ('word', 'char'), 'Unknown token type: ' + token
    return [line.split() if token == 'word' else list(line) for line in lines]
```

```
#@save
class MaskedSoftmaxCELoss(gluon.loss.SoftmaxCELoss):
    """ マスク付きの softmax クロスエントロピー損失。"""
    # `pred` shape: (`batch_size`, `num_steps`, `vocab_size`)
    # `label` shape: (`batch_size`, `num_steps`)
    # `valid_len` shape: (`batch_size`,)
    def forward(self, pred, label, valid_len):
        # `weights` shape: (`batch_size`, `num_steps`, 1)
        weights = np.expand_dims(np.ones_like(label), axis=-1)
        weights = npx.sequence_mask(weights, valid_len, True, axis=1)
        return super(MaskedSoftmaxCELoss, self).forward(pred, label, weights)

#@save
def train_seq2seq(net, data_iter, lr, num_epochs, tgt_vocab, device):
```

(continues on next page)

(continued from previous page)

```
    """ 系列から系列へのモデルを学習する。"""
    net.initialize(init.Xavier(), force_reinit=True, ctx=device)
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    loss = MaskedSoftmaxCELoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs])
    for epoch in range(num_epochs):
        timer = d2l.Timer()
        metric = d2l.Accumulator(2)  # 学習損失の合計、トークン数
        for batch in data_iter:
            X, X_valid_len, Y, Y_valid_len = [
                x.as_in_ctx(device) for x in batch]
            bos = np.array(
                [tgt_vocab['<bos>']] * Y.shape[0], ctx=device).reshape(-1, 1)
            dec_input = d2l.concat([bos, Y[:, :-1]], 1)  # Teacher forcing
            with autograd.record():
                Y_hat, _ = net(X, dec_input, X_valid_len)
                l = loss(Y_hat, Y, Y_valid_len)
            l.backward()
            d2l.grad_clipping(net, 1)
            num_tokens = Y_valid_len.sum()
            trainer.step(num_tokens)
            metric.add(l.sum(), num_tokens)
        if (epoch + 1) % 10 == 0:
            animator.add(epoch + 1, (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, {metric[1] / timer.stop():.1f} '
          f'tokens/sec on {str(device)}')

#@save
def predict_seq2seq(net, src_sentence, src_vocab, tgt_vocab, num_steps,
                    device, save_attention_weights=False):
    """ 系列から系列への予測を行う。"""
    src_tokens = src_vocab[src_sentence.lower().split(' ')] + [
        src_vocab['<eos>']]
    enc_valid_len = np.array([len(src_tokens)], ctx=device)
    src_tokens = d2l.truncate_pad(src_tokens, num_steps, src_vocab['<pad>'])
    # バッチ次元を追加する
    enc_X = np.expand_dims(np.array(src_tokens, ctx=device), axis=0)
    enc_outputs = net.encoder(enc_X, enc_valid_len)
    dec_state = net.decoder.init_state(enc_outputs, enc_valid_len)
    # バッチ次元を追加する
    dec_X = np.expand_dims(np.array([tgt_vocab['<bos>']], ctx=device), axis=0)
    output_seq, attention_weight_seq = [], []
    for _ in range(num_steps):
        Y, dec_state = net.decoder(dec_X, dec_state)
        # 次時刻のデコーダ入力として、予測確率が最も高いトークンを使う
        dec_X = Y.argmax(axis=2)
        pred = dec_X.squeeze(axis=0).astype('int32').item()
        # 注意重みを保存する（後で扱う）
        if save_attention_weights:
            attention_weight_seq.append(net.decoder.attention_weights)
        # 終了トークンが予測されたら、出力系列の生成は完了
        if pred == tgt_vocab['<eos>']:
            break
```

(continues on next page)

(continued from previous page)

```
        output_seq.append(pred)
    return ' '.join(tgt_vocab.to_tokens(output_seq)), attention_weight_seq
```

```
#@save
d2l.DATA_HUB['fra-eng'] = (d2l.DATA_URL + 'fra-eng.zip',
                           '94646ad1522d915e7b0f9296181140edcf86a4f5')

#@save
def read_data_nmt():
    """ 英仏データセットを読み込む。"""
    data_dir = d2l.download_extract('fra-eng')
    with open(os.path.join(data_dir, 'fra.txt'), 'r', encoding='utf-8') as f:
        return f.read()

#@save
def preprocess_nmt(text):
    """ 英仏データセットを前処理する。"""
    def no_space(char, prev_char):
        return char in set(',.!?') and prev_char != ' '

    # 改行なしスペースをスペースに置換し、大文字を小文字に変換する
    text = text.replace('\u202f', ' ').replace('\xa0', ' ').lower()
    # 単語と句読点の間にスペースを挿入する
    out = [' ' + char if i > 0 and no_space(char, text[i - 1]) else char
           for i, char in enumerate(text)]
    return ''.join(out)

#@save
def tokenize_nmt(text, num_examples=None):
    """ 英仏データセットをトークン化する。"""
    source, target = [], []
    for i, line in enumerate(text.split('\n')):
        if num_examples and i > num_examples:
            break
        parts = line.split('\t')
        if len(parts) == 2:
            source.append(parts[0].split(' '))
            target.append(parts[1].split(' '))
    return source, target


#@save
def truncate_pad(line, num_steps, padding_token):
    """ 系列を切り詰めるかパディングする。"""
    if len(line) > num_steps:
        return line[:num_steps]  # 切り詰め
    return line + [padding_token] * (num_steps - len(line))  # パディング


#@save
def build_array_nmt(lines, vocab, num_steps):
    """ 機械翻訳のテキスト系列をミニバッチに変換する。"""
    lines = [vocab[l] for l in lines]
```

(continues on next page)

(continued from previous page)

```
    lines = [l + [vocab['<eos>']] for l in lines]
    array = d2l.tensor([truncate_pad(
        l, num_steps, vocab['<pad>']) for l in lines])
    valid_len = d2l.reduce_sum(
        d2l.astype(array != vocab['<pad>'], d2l.int32), 1)
    return array, valid_len


#@save
def load_data_nmt(batch_size, num_steps, num_examples=600):
    """ 翻訳データセットのイテレータと語彙を返す。"""
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)
    src_vocab = d2l.Vocab(source, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    tgt_vocab = d2l.Vocab(target, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    src_array, src_valid_len = build_array_nmt(source, src_vocab, num_steps)
    tgt_array, tgt_valid_len = build_array_nmt(target, tgt_vocab, num_steps)
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = d2l.load_array(data_arrays, batch_size)
    return data_iter, src_vocab, tgt_vocab
```

```
def evaluate_loss(net, data_iter, loss):  #@save
    """ 与えられたデータセット上でモデルの損失を評価する。"""
    metric = d2l.Accumulator(2)  # 損失の合計、サンプル数
    for X, y in data_iter:
        out = net(X)
        y = d2l.reshape(y, out.shape)
        l = loss(out, y)
        metric.add(d2l.reduce_sum(l), d2l.size(l))
    return metric[0] / metric[1]
```

```
#@save
def sequence_mask(X, valid_len, value=0):
    """ 系列中の不要な要素をマスクする。"""
    maxlen = X.shape[1]
    mask = tf.range(start=0, limit=maxlen, dtype=tf.float32)[
        None, :] < tf.cast(valid_len[:, None], dtype=tf.float32)

    if len(X.shape) == 3:
        return tf.where(tf.expand_dims(mask, axis=-1), X, value)
    else:
        return tf.where(mask, X, value)


#@save
class MaskedSoftmaxCELoss(tf.keras.losses.Loss):
    """ マスク付きの softmax クロスエントロピー損失。"""
    def __init__(self, valid_len):
        super().__init__(reduction='none')
        self.valid_len = valid_len
```

(continues on next page)

(continued from previous page)

```
    # `pred` shape: (`batch_size`, `num_steps`, `vocab_size`)
    # `label` shape: (`batch_size`, `num_steps`)
    # `valid_len` shape: (`batch_size`,)
    def call(self, label, pred):
        weights = tf.ones_like(label, dtype=tf.float32)
        weights = sequence_mask(weights, self.valid_len)
        label_one_hot = tf.one_hot(label, depth=pred.shape[-1])
        unweighted_loss = tf.keras.losses.CategoricalCrossentropy(
            from_logits=True, reduction='none')(label_one_hot, pred)
        weighted_loss = tf.reduce_mean((unweighted_loss*weights), axis=1)
        return weighted_loss

#@save
def train_seq2seq(net, data_iter, lr, num_epochs, tgt_vocab, device):
    """ 系列から系列へのモデルを学習する。"""
    optimizer = tf.keras.optimizers.Adam(learning_rate=lr)
    animator = d2l.Animator(xlabel="epoch", ylabel="loss",
                            xlim=[10, num_epochs])
    for epoch in range(num_epochs):
        timer = d2l.Timer()
        metric = d2l.Accumulator(2)  # 学習損失の合計、トークン数
        for batch in data_iter:
            X, X_valid_len, Y, Y_valid_len = [x for x in batch]
            bos = tf.reshape(tf.constant([tgt_vocab['<bos>']] * Y.shape[0]),
                             shape=(-1, 1))
            dec_input = tf.concat([bos, Y[:, :-1]], 1)  # Teacher forcing
            with tf.GradientTape() as tape:
                Y_hat, _ = net(X, dec_input, X_valid_len, training=True)
                l = MaskedSoftmaxCELoss(Y_valid_len)(Y, Y_hat)
            gradients = tape.gradient(l, net.trainable_variables)
            gradients = d2l.grad_clipping(gradients, 1)
            optimizer.apply_gradients(zip(gradients, net.trainable_variables))
            num_tokens = tf.reduce_sum(Y_valid_len).numpy()
            metric.add(tf.reduce_sum(l), num_tokens)
        if (epoch + 1) % 10 == 0:
            animator.add(epoch + 1, (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, {metric[1] / timer.stop():.1f} '
          f'tokens/sec on {str(device._device_name)}')

#@save
def predict_seq2seq(net, src_sentence, src_vocab, tgt_vocab, num_steps,
                    save_attention_weights=False):
    """ 系列から系列への予測を行う。"""
    src_tokens = src_vocab[src_sentence.lower().split(' ')] + [
        src_vocab['<eos>']]
    enc_valid_len = tf.constant([len(src_tokens)])
    src_tokens = d2l.truncate_pad(src_tokens, num_steps, src_vocab['<pad>'])
    # バッチ次元を追加する
    enc_X = tf.expand_dims(src_tokens, axis=0)
    enc_outputs = net.encoder(enc_X, enc_valid_len, training=False)
    dec_state = net.decoder.init_state(enc_outputs, enc_valid_len)
    # バッチ次元を追加する
    dec_X = tf.expand_dims(tf.constant([tgt_vocab['<bos>']]), axis=0)
    output_seq, attention_weight_seq = [], []
```

(continues on next page)

(continued from previous page)

```
    for _ in range(num_steps):
        Y, dec_state = net.decoder(dec_X, dec_state, training=False)
        # 次時刻のデコーダ入力として、予測確率が最も高いトークンを使う
        dec_X = tf.argmax(Y, axis=2)
        pred = tf.squeeze(dec_X, axis=0)
        # 注意重みを保存する
        if save_attention_weights:
            attention_weight_seq.append(net.decoder.attention_weights)
        # 終了トークンが予測されたら、出力系列の生成は完了
        if pred == tgt_vocab['<eos>']:
            break
        output_seq.append(pred.numpy())
    return ' '.join(tgt_vocab.to_tokens(tf.reshape(output_seq, shape = -1).
↪numpy().tolist())), attention_weight_seq
```

```
def grad_clipping(net, theta):  #@save
    """ 勾配をクリップする。"""
    if isinstance(net, nn.Module):
        params = [p for p in net.parameters() if p.requires_grad]
    else:
        params = net.params
    norm = torch.sqrt(sum(torch.sum((p.grad ** 2)) for p in params))
    if norm > theta:
        for param in params:
            param.grad[:] *= theta / norm
```

注意機構の章に関する追加。

```
#@save
d2l.DATA_HUB['fra-eng'] = (d2l.DATA_URL + 'fra-eng.zip',
                           '94646ad1522d915e7b0f9296181140edcf86a4f5')

#@save
def read_data_nmt():
    """ 英仏データセットを読み込む。"""
    data_dir = d2l.download_extract('fra-eng')
    with open(os.path.join(data_dir, 'fra.txt'), 'r', encoding='utf-8') as f:
        return f.read()

#@save
def preprocess_nmt(text):
    """ 英仏データセットを前処理する。"""
    def no_space(char, prev_char):
        return char in set(',.!?') and prev_char != ' '

    # 改行なしスペースをスペースに置換し、大文字を小文字に変換する
    text = text.replace('\u202f', ' ').replace('\xa0', ' ').lower()
    # 単語と句読点の間にスペースを挿入する
    out = [' ' + char if i > 0 and no_space(char, text[i - 1]) else char
           for i, char in enumerate(text)]
    return ''.join(out)
```

(continues on next page)

(continued from previous page)

```
#@save
def tokenize_nmt(text, num_examples=None):
    """ 英仏データセットをトークン化する。"""
    source, target = [], []
    for i, line in enumerate(text.split('\n')):
        if num_examples and i > num_examples:
            break
        parts = line.split('\t')
        if len(parts) == 2:
            source.append(parts[0].split(' '))
            target.append(parts[1].split(' '))
    return source, target


#@save
def truncate_pad(line, num_steps, padding_token):
    """ 系列を切り詰めるかパディングする。"""
    if len(line) > num_steps:
        return line[:num_steps]  # 切り詰め
    return line + [padding_token] * (num_steps - len(line))  # パディング


#@save
def build_array_nmt(lines, vocab, num_steps):
    """ 機械翻訳のテキスト系列をミニバッチに変換する。"""
    lines = [vocab[l] for l in lines]
    lines = [l + [vocab['<eos>']] for l in lines]
    array = d2l.tensor([truncate_pad(
        l, num_steps, vocab['<pad>']) for l in lines])
    valid_len = d2l.reduce_sum(
        d2l.astype(array != vocab['<pad>'], d2l.int32), 1)
    return array, valid_len


#@save
def load_data_nmt(batch_size, num_steps, num_examples=600):
    """ 翻訳データセットのイテレータと語彙を返す。"""
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)
    src_vocab = d2l.Vocab(source, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    tgt_vocab = d2l.Vocab(target, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    src_array, src_valid_len = build_array_nmt(source, src_vocab, num_steps)
    tgt_array, tgt_valid_len = build_array_nmt(target, tgt_vocab, num_steps)
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = d2l.load_array(data_arrays, batch_size)
    return data_iter, src_vocab, tgt_vocab
```

```
#@save
def sequence_mask(X, valid_len, value=0):
    """ 系列中の不要な要素をマスクする。"""
    maxlen = X.size(1)
```

(continues on next page)

(continued from previous page)

```
    mask = torch.arange((maxlen), dtype=torch.float32,
                        device=X.device)[None, :] < valid_len[:, None]
    X[~mask] = value
    return X


#@save
class MaskedSoftmaxCELoss(nn.CrossEntropyLoss):
    """ マスク付きの softmax クロスエントロピー損失。"""
    # `pred` shape: (`batch_size`, `num_steps`, `vocab_size`)
    # `label` shape: (`batch_size`, `num_steps`)
    # `valid_len` shape: (`batch_size`,)
    def forward(self, pred, label, valid_len):
        weights = torch.ones_like(label)
        weights = sequence_mask(weights, valid_len)
        self.reduction='none'
        unweighted_loss = super(MaskedSoftmaxCELoss, self).forward(
            pred.permute(0, 2, 1), label)
        weighted_loss = (unweighted_loss * weights).mean(dim=1)
        return weighted_loss

#@save
def train_seq2seq(net, data_iter, lr, num_epochs, tgt_vocab, device):
    """ 系列から系列へのモデルを学習する。"""
    def xavier_init_weights(m):
        if type(m) == nn.Linear:
            nn.init.xavier_uniform_(m.weight)
        if type(m) == nn.GRU:
            for param in m._flat_weights_names:
                if "weight" in param:
                    nn.init.xavier_uniform_(m._parameters[param])
    net.apply(xavier_init_weights)
    net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    loss = MaskedSoftmaxCELoss()
    net.train()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs])
    for epoch in range(num_epochs):
        timer = d2l.Timer()
        metric = d2l.Accumulator(2)  # 学習損失の合計、トークン数
        for batch in data_iter:
            optimizer.zero_grad()
            X, X_valid_len, Y, Y_valid_len = [x.to(device) for x in batch]
            bos = torch.tensor([tgt_vocab['<bos>']] * Y.shape[0],
                               device=device).reshape(-1, 1)
            dec_input = d2l.concat([bos, Y[:, :-1]], 1)  # Teacher forcing
            Y_hat, _ = net(X, dec_input, X_valid_len)
            l = loss(Y_hat, Y, Y_valid_len)
            l.sum().backward()  # `backward` のために損失をスカラーにする
            d2l.grad_clipping(net, 1)
            num_tokens = Y_valid_len.sum()
            optimizer.step()
            with torch.no_grad():
                metric.add(l.sum(), num_tokens)
```

(continues on next page)

(continued from previous page)

```
        if (epoch + 1) % 10 == 0:
            animator.add(epoch + 1, (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, {metric[1] / timer.stop():.1f} '
          f'tokens/sec on {str(device)}')


#@save
def predict_seq2seq(net, src_sentence, src_vocab, tgt_vocab, num_steps,
                    device, save_attention_weights=False):
    """ 系列から系列への予測を行う。"""
    # 推論のために `net` を評価モードに設定する
    net.eval()
    src_tokens = src_vocab[src_sentence.lower().split(' ')] + [
        src_vocab['<eos>']]
    enc_valid_len = torch.tensor([len(src_tokens)], device=device)
    src_tokens = d2l.truncate_pad(src_tokens, num_steps, src_vocab['<pad>'])
    # バッチ次元を追加する
    enc_X = torch.unsqueeze(
        torch.tensor(src_tokens, dtype=torch.long, device=device), dim=0)
    enc_outputs = net.encoder(enc_X, enc_valid_len)
    dec_state = net.decoder.init_state(enc_outputs, enc_valid_len)
    # バッチ次元を追加する
    dec_X = torch.unsqueeze(torch.tensor(
        [tgt_vocab['<bos>']], dtype=torch.long, device=device), dim=0)
    output_seq, attention_weight_seq = [], []
    for _ in range(num_steps):
        Y, dec_state = net.decoder(dec_X, dec_state)
        # 次時刻のデコーダ入力として、予測確率が最も高いトークンを使う
        dec_X = Y.argmax(dim=2)
        pred = dec_X.squeeze(dim=0).type(torch.int32).item()
        # 注意重みを保存する（後で扱う）
        if save_attention_weights:
            attention_weight_seq.append(net.decoder.attention_weights)
        # 終了トークンが予測されたら、出力系列の生成は完了
        if pred == tgt_vocab['<eos>']:
            break
        output_seq.append(pred)
    return ' '.join(tgt_vocab.to_tokens(output_seq)), attention_weight_seq
```

26.8 d2l API ドキュメント

172



この節では、d2l パッケージに含まれるクラスと関数をアルファベット順に示し、それらが本書のどこで定義されているかを示す。これにより、より詳細な実装や説明を見つけることができる。GitHub リポジトリ[172] のソースコードも参照のこと。

173



完全な API リファレンス（自動生成されたクラスおよび関数のドキュメント）は、英語版の d2l.ai API Document[173] を参照のこと。

26.8.1 クラス

クラス名	説明
AdditiveAttention	加法注意機構
AddNorm	残差接続と層正規化
AttentionDecoder	注意機構付きデコーダの基底クラス
Classifier	分類モデルの基底クラス
DataModule	データモジュールの基底クラス
Decoder	デコーダの基底クラス
DotProductAttention	スケーリングドット積注意機構
Encoder	エンコーダの基底クラス
EncoderDecoder	エンコーダ・デコーダモデルの基底クラス
FashionMNIST	Fashion-MNIST データセット
GRU	ゲート付き回帰ユニット
HyperParameters	ハイパーパラメータのユーティリティ基底クラス
LeNet	LeNet-5 モデル
LinearRegression	線形回帰モデル
LinearRegressionScratch	ゼロから実装した線形回帰
Module	すべてのモデルの基底クラス
MTFraEng	仏英機械翻訳データセット
MultiHeadAttention	マルチヘッド注意機構
PositionalEncoding	位置エンコーディング
PositionWiseFFN	位置ごとのフィードフォワードネットワーク
ProgressBoard	学習進捗の可視化ボード
Residual	残差ブロック
ResNeXtBlock	ResNeXt ブロック
RNN	再帰ニューラルネットワーク
RNNLM	RNN 言語モデル
RNNLMScratch	ゼロから実装した RNN 言語モデル
RNNScratch	ゼロから実装した RNN
Seq2Seq	シーケンスツーシーケンスモデル
Seq2SeqEncoder	Seq2Seq エンコーダ
SGD	確率的勾配降下法
SoftmaxRegression	ソフトマックス回帰モデル
SyntheticRegressionData	合成回帰データ
TimeMachine	Time Machine テキストデータセット
Trainer	モデルの学習ユーティリティ
TransformerEncoder	Transformer エンコーダ
TransformerEncoderBlock	Transformer エンコーダブロック
Vocab	テキストの語彙

26.8.2 関数

関数名	説明
add_to_class	クラスにメソッドを動的に追加
bleu	BLEU スコアの計算
check_len	引数の長さを検証
check_shape	テンソルの形状を検証
corr2d	2 次元相互相関演算
cpu	CPU デバイスを取得
gpu	GPU デバイスを取得
init_cnn	CNN の重みを初期化
init_seq2seq	Seq2Seq の重みを初期化
masked_softmax	マスク付きソフトマックス演算
num_gpus	利用可能な GPU 数を取得
plot	グラフの描画
set_axes	matplotlib の軸を設定
set_figsize	図のサイズを設定
show_heatmaps	ヒートマップを表示
show_list_len_pair_hist	リスト長のペアのヒストグラムを表示
try_all_gpus	利用可能な全 GPU を取得、なければ CPU
try_gpu	GPU を取得、なければ CPU
use_svg_display	SVG 形式での表示を有効化

参考文献 ==========

Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ⋯et al. (2016). TensorFlow: a system for large-scale machine learning. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (pp. 265–283).

Abdel-Hamid, O., Mohamed, A.-R., Jiang, H., Deng, L., Penn, G., & Yu, D. (2014). Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, *22*(10), 1533–1545.

Ahmed, A., Aly, M., Gonzalez, J., Narayanamurthy, S., & Smola, A. J. (2012). Scalable inference in latent variable models. *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining* (pp. 123–132).

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: a next-generation hyperparameter optimization framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

Alayrac, J.-B., Donahue, J., Luc, P., Miech, A., Barr, I., Hasson, Y., ⋯et al. (2022). Flamingo: a visual language model for few-shot learning. *ArXiv:2204.14198*.

Alsallakh, B., Kokhlikyan, N., Miglani, V., Yuan, J., & Reblitz-Richardson, O. (2020). Mind the PAD –CNNs can develop blind spots. *ArXiv:2010.02178*.

Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., ⋯et al. (2023). PaLM 2 Technical Report. *ArXiv:2305.10403*.

Anil, R., Gupta, V., Koren, T., Regan, K., & Singer, Y. (2020). Scalable second-order optimization for deep learning. *ArXiv:2002.09018*.

Aronszajn, N. (1950). Theory of reproducing kernels. *Transactions of the American Mathematical Society*, *68*(3), 337–404.

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *ArXiv:1607.06450*.

Baevski, A., & Auli, M. (2018). Adaptive input representations for neural language modeling. *International Conference on Learning Representations*.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *ArXiv:1409.0473*.

Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., ⋯et al. (2022). Constitutional AI: harmlessness from AI feedback. *ArXiv:2212.08073*.

Baptista, R., & Poloczek, M. (2018). Bayesian optimization of combinatorial structures. *Proceedings of the 35th International Conference on Machine Learning*.

Bardenet, R., Brendel, M., Kégl, B., & Sebag, M. (2013). Collaborative hyperparameter tuning. *Proceedings of the 30th International Conference on Machine Learning (ICML'13)*.

Bay, H., Tuytelaars, T., & Van Gool, L. (2006). SURF: Speeded up robust features. *European Conference on Computer Vision* (pp. 404–417).

Bellman, R. (1966). Dynamic programming. *Science*, *153*, 34–37.

Bellman, R. (1952). On the theory of dynamic programming. *Proceedings of the National Academy of Sciences*, *38*(8), 716–719.

Bellman, R. (1957). A Markovian decision process. *Journal of Mathematics and Mechanics*, *6*(5), 679–684. URL: http://www.jstor.org/stable/24900506

Bellman, R. (1957). *Dynamic Programming*. Dover Publications.

Beltagy, I., Peters, M. E., & Cohan, A. (2020). Longformer: the long-document transformer. *ArXiv:2004.05150*.

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, *3*(Feb), 1137–1155.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157–166.

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. *Advances in Neural Information Processing Systems*, *24*.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., ⋯Bengio, Y. (2010). Theano: a CPU and GPU math compiler in Python. *Proc. 9th Python in Science Conference* (pp. 3–10).

Beutel, A., Murray, K., Faloutsos, C., & Smola, A. J. (2014). CoBaFi: collaborative Bayesian filtering. *Proceedings of the 23rd International Conference on World Wide Web* (pp. 97–108).

Bishop, C. M. (1995). Training with noise is equivalent to Tikhonov regularization. *Neural Computation*, *7*(1), 108–116.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Black, F., & Scholes, M. (1973). The pricing of options and corporate liabilities. *Journal of Political Economy*, *81*, 637–654.

Bodla, N., Singh, B., Chellappa, R., & Davis, L. S. (2017). Soft-NMS-improving object detection with one line of code. *Proceedings of the IEEE International Conference on Computer Vision* (pp. 5561–5569).

Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, *5*, 135–146.

Bollobás, B. (1999). *Linear Analysis*. Cambridge University Press.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., ⋯et al. (2021). On the opportunities and risks of foundation models. *ArXiv:2108.07258*.

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. *Proceedings of COMPSTAT'2010* (pp. 177–186). Springer.

Bottou, L., & Le Cun, Y. (1988). SN: a simulator for connectionist models. *Proceedings of NeuroNimes 88* (pp. 371–382). Nimes, France. URL: http://leon.bottou.org/papers/bottou-lecun-88

Boucheron, S., Bousquet, O., & Lugosi, G. (2005). Theory of classification: a survey of some recent advances. *ESAIM: Probability and Statistics*, *9*, 323–375.

Bowman, S. R., Angeli, G., Potts, C., & Manning, C. D. (2015). A large annotated corpus for learning natural language inference. *ArXiv:1508.05326*.

Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization*. Cambridge, England: Cambridge University Press.

Bradley, R. A., & Terry, M. E. (1952). Rank analysis of incomplete block designs: I. The method of paired comparisons. *Biometrika*, *39*(3/4), 324–345.

Brown, N., & Sandholm, T. (2017). Libratus: the superhuman AI for no-limit poker. *IJCAI* (pp. 5226–5228).

Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Lafferty, J., ⋯Roossin, P. S. (1990). A statistical approach to machine translation. *Computational Linguistics*, *16*(2), 79–85.

Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Mercer, R. L., & Roossin, P. (1988). A statistical approach to language translation. *COLING Budapest 1988 Volume 1: International Conference on Computational Linguistics*.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ⋯et al. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, *33*, 1877–1901.

Buslaev, A., Iglovikov, V. I., Khvedchenya, E., Parinov, A., Druzhinin, M., & Kalinin, A. A. (2020). Albumentations: Fast and flexible image augmentations. *Information*, *11*(2), 125.

Campbell, M., Hoane Jr, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial Intelligence*, *134*(1-2), 57–83.

Canny, J. (1987). A computational approach to edge detection. *Readings in Computer Vision* (pp. 184–203). Elsevier.

Cer, D., Diab, M., Agirre, E., Lopez-Gazpio, I., & Specia, L. (2017). SemEval-2017 Task 1: semantic textual similarity multilingual and crosslingual focused evaluation. *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)* (pp. 1–14).

Chan, W., Jaitly, N., Le, Q. V., & Vinyals, O. (2015). Listen, attend and spell. *ArXiv:1508.01211*.

Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., ⋯Mordatch, I. (2021). Decision transformer: reinforcement learning via sequence modeling. *Advances in Neural Information Processing Systems*, *34*, 15084–15097.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., ⋯Zhang, Z. (2015). MXNET: a flexible and efficient machine learning library for heterogeneous distributed systems. *ArXiv:1512.01274*.

Cheng, J., Dong, L., & Lapata, M. (2016). Long short-term memory-networks for machine reading. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing* (pp. 551–561).

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., & Shelhamer, E. (2014). CuDNN: Efficient primitives for deep learning. *ArXiv:1410.0759*.

Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder–decoder approaches. *ArXiv:1409.1259*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. *ArXiv:1406.1078*.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., ⋯et al. (2022). PaLM: scaling language modeling with pathways. *ArXiv:2204.02311*.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *ArXiv:1412.3555*.

Clark, K., Luong, M.-T., Le, Q. V., & Manning, C. D. (2020). ELECTRA: pre-training text encoders as discriminators rather than generators. *International Conference on Learning Representations*.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, *12*, 2493–2537.

Cordonnier, J.-B., Loukas, A., & Jaggi, M. (2020). On the relationship between self-attention and convolutional layers. *International Conference on Learning Representations*.

Cover, T., & Thomas, J. (1999). *Elements of Information Theory*. John Wiley & Sons.

Csiszár, I. (2008). Axiomatic characterizations of information measures. *Entropy*, *10*(3), 261–273.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*(4), 303–314.

Dalal, N., & Triggs, B. (2005). Histograms of oriented gradients for human detection. *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)* (pp. 886–893).

De Cock, D. (2011). Ames, Iowa: alternative to the Boston housing data as an end of semester regression project. *Journal of Statistics Education*, *19*(3).

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., ⋯et al. (2012). Large scale distributed deep networks. *Proceedings of the 25th International Conference on Neural Information Processing Systems, Volume 1* (pp. 1223–1231).

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ⋯ Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review* (pp. 205–220).

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). Imagenet: a large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition* (pp. 248–255).

Der Kiureghian, A., & Ditlevsen, O. (2009). Aleatory or epistemic? does it matter? *Structural Safety*, *31*(2), 105–112.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of deep bidirectional transformers for language understanding. *ArXiv:1810.04805*.

Doersch, C., Gupta, A., & Efros, A. A. (2015). Unsupervised visual representation learning by context prediction. *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1422–1430).

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ⋯ et al. (2021). An image is worth 16 x 16 words: transformers for image recognition at scale. *International Conference on Learning Representations*.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*, 2121–2159.

Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. *ArXiv:1603.07285*.

Dwivedi, V. P., & Bresson, X. (2020). A generalization of transformer networks to graphs. *ArXiv:2012.09699*.

Dwork, C., Feldman, V., Hardt, M., Pitassi, T., Reingold, O., & Roth, A. L. (2015). Preserving statistical validity in adaptive data analysis. *Proceedings of the 47th Annual ACM Symposium on Theory of Computing* (pp. 117–126).

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, *14*(2), 179–211.

Elsken, T., Metzen, J. H., & Hutter, F. (2018). Neural architecture search: a ssurvey. *ArXiv:1808.05377 [stat.ML]*.

Fechner, G. T. (1860). *Elemente der Psychophysik*. Vol. 2. Breitkopf u. Härtel.

Fedus, W., Zoph, B., & Shazeer, N. (2022). Switch transformers: scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, *23*(120), 1–39.

Fernando, R. (2004). *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley.

Feurer, M., & Hutter, F. (2018). Hyperparameter ptimization. *Automatic Machine Learning: Methods, Systems, Challenges*. Springer.

Feurer, M., Letham, B., Hutter, F., & Bakshy, E. (2022). Practical transfer learning for Bayesian optimization. *ArXiv:1802.02219 [stat.ML]*.

Field, D. J. (1987). Relations between the statistics of natural images and the response properties of cortical cells. *JOSA A*, *4*(12), 2379–2394.

Fisher, R. A. (1925). *Statistical Methods for Research Workers.* Oliver & Boyd.

Flammarion, N., & Bach, F. (2015). From averaging to acceleration, there is only a stepsize. *Conference on Learning Theory* (pp. 658–695).

Forrester, A. I., Sóbester, A., & Keane, A. J. (2007). Multi-fidelity optimization via surrogate modelling. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *463*(2088), 3251–3269.

Franceschi, L., Donini, M., Frasconi, P., & Pontil, M. (2017). Forward and reverse gradient-based hyperparameter optimization. *Proceedings of the 34th International Conference on Machine Learning (ICML'17)*.

Frankle, J., & Carbin, M. (2018). The lottery ticket hypothesis: finding sparse, trainable neural networks. *ArXiv:1803.03635*.

Frazier, P. I. (2018). A tutorial on Bayesian optimization. *ArXiv:1807.02811*.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. *Proceedings of the International Conference on Machine Learning* (pp. 148–156).

Friedman, J. H. (1987). Exploratory projection pursuit. *Journal of the American Statistical Association*, *82*(397), 249–266.

Frostig, R., Johnson, M. J., & Leary, C. (2018). Compiling machine learning programs via high-level tracing. *Proceedings of Systems for Machine Learning*.

Fukushima, K. (1982). Neocognitron: a self-organizing neural network model for a mechanism of visual pattern recognition. *Competition and Cooperation in Neural Nets* (pp. 267–285). Springer.

Gardner, J., Pleiss, G., Weinberger, K. Q., Bindel, D., & Wilson, A. G. (2018). GPyTorch: blackbox matrix–matrix Gaussian process inference with GPU acceleration. *Advances in Neural Information Processing Systems*.

Garg, S., Balakrishnan, S., Kolter, Z., & Lipton, Z. (2021). RATT: leveraging unlabeled data to guarantee generalization. *International Conference on Machine Learning* (pp. 3598–3609).

Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). Image style transfer using convolutional neural networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2414–2423).

Gauss, C. F. (1809). Theoria motus corporum coelestum. *Werke*. Königlich Preussische Akademie der Wissenschaften.

Gibbs, J. W. (1902). *Elementary Principles of Statistical Mhanics*. Scribner's.

Ginibre, J. (1965). Statistical ensembles of complex, quaternion, and real matrices. *Journal of Mathematical Physics*, *6*(3), 440–449.

Girshick, R. (2015). Fast R-CNN. *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1440–1448).

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 580–587).

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (pp. 249–256).

Goh, G. (2017). Why momentum really works. *Distill*. URL: http://distill.pub/2017/momentum

Goldberg, D., Nichols, D., Oki, B. M., & Terry, D. (1992). Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, *35*(12), 61–71.

Golub, G. H., & Van Loan, C. F. (1996). *Matrix Computations*. Johns Hopkins University Press.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ⋯Bengio, Y. (2014). Generative adversarial nets. *Advances in Neural Information Processing Systems* (pp. 2672–2680).

Gotmare, A., Keskar, N. S., Xiong, C., & Socher, R. (2018). A closer look at deep learning heuristics: learning rate restarts, warmup and distillation. *ArXiv:1810.13243*.

Goyal, A., Bochkovskiy, A., Deng, J., & Koltun, V. (2021). Non-deep networks. *ArXiv:2110.07641*.

Graham, B. (2014). Fractional max-pooling. *ArXiv:1412.6071*.

Graves, A. (2013). Generating sequences with recurrent neural networks. *ArXiv:1308.0850*.

Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H., & Schmidhuber, J. (2008). A novel connectionist system for unconstrained handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *31*(5), 855–868.

Graves, A., & Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, *18*(5-6), 602–610.

Griewank, A. (1989). On automatic differentiation. *Mathematical Programming: Recent Developments and Applications* (pp. 83–107). Kluwer.

Gulati, A., Qin, J., Chiu, C.-C., Parmar, N., Zhang, Y., Yu, J., ⋯et al. (2020). Conformer: convolution-augmented transformer for speech recognition. *Proc. Interspeech 2020*, pp. 5036–5040.

Gunawardana, A., & Shani, G. (2015). Evaluating recommender systems. *Recommender Systems Handbook* (pp. 265–308). Springer.

Guo, H., Tang, R., Ye, Y., Li, Z., & He, X. (2017). Deepfm: a factorization-machine based neural network for ctr prediction. *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (pp. 1725–1731).

Guyon, I., Gunn, S., Nikravesh, M., & Zadeh, L. A. (2008). *Feature Extraction: Foundations and Applications*. Springer.

Hadjis, S., Zhang, C., Mitliagkas, I., Iter, D., & Ré, C. (2016). Omnivore: an optimizer for multi-device deep learning on CPUs and GPUs. *ArXiv:1606.04487*.

Hartley, R., & Zisserman, A. (2000). *Multiple View Geometry in Computer Vision*. Cambridge University Press.

Hartley, R. I., & Kahl, F. (2009). Global optimization through rotation space search. *International Journal of Computer Vision*, *82*(1), 64–79.

He, K., Chen, X., Xie, S., Li, Y., Dollár, P., & Girshick, R. (2022). Masked autoencoders are scalable vision learners. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 16000–16009).

He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask R-CNN. *Proceedings of the IEEE International Conference on Computer Vision* (pp. 2961–2969).

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1026–1034).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Identity mappings in deep residual networks. *European Conference on Computer Vision* (pp. 630–645).

He, X., & Chua, T.-S. (2017). Neural factorization machines for sparse predictive analytics. *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 355–364).

He, X., Liao, L., Zhang, H., Nie, L., Hu, X., & Chua, T.-S. (2017). Neural collaborative filtering. *Proceedings of the 26th International Conference on World Wide Web* (pp. 173–182).

Hebb, D. O. (1949). *The Organization of Behavior*. Wiley.

Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (GELUs). *ArXiv:1606.08415*.

Hennessy, J. L., & Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*. Elsevier.

Herlocker, J. L., Konstan, J. A., Borchers, A., & Riedl, J. (1999). An algorithmic framework for performing collaborative filtering. *22nd Annual International ACM Conference on Research and Development in Information Retrieval, SIGIR 1999* (pp. 230–237).

Hidasi, B., Karatzoglou, A., Baltrunas, L., & Tikk, D. (2015). Session-based recommendations with recurrent neural networks. *ArXiv:1511.06939*.

Ho, J., Jain, A., & Abbeel, P. (2020). Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, *33*, 6840–6851.

Hochreiter, S., Bengio, Y., Frasconi, P., & Schmidhuber, J. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., ⋯et al. (2022). Training compute-optimal large language models. *ArXiv:2203.15556*.

Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., ⋯Adam, H. (2019). Searching for MobileNetV3. *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 1314–1324).

Hoyer, P. O., Janzing, D., Mooij, J. M., Peters, J., & Schölkopf, B. (2009). Nonlinear causal discovery with additive noise models. *Advances in Neural Information Processing Systems* (pp. 689–696).

Hu, J., Shen, L., & Sun, G. (2018). Squeeze-and-excitation networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 7132–7141).

Hu, Y., Koren, Y., & Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets. *2008 8th IEEE International Conference on Data Mining* (pp. 263–272).

Hu, Z., Lee, R. K.-W., Aggarwal, C. C., & Zhang, A. (2022). Text style transfer: a review and experimental evaluation. *SIGKDD Explor. Newsl.*, *24*(1). URL: https://doi.org/10.1145/3544903.3544906

Huang, C.-Z. A., Vaswani, A., Uszkoreit, J., Simon, I., Hawthorne, C., Shazeer, N., ⋯Eck, D. (2018). Music transformer: generating music with long-term structure. *International Conference on Learning Representations*.

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4700–4708).

Huang, Z., Xu, W., & Yu, K. (2015). Bidirectional LSTM–CRF models for sequence tagging. *ArXiv:1508.01991*.

Hubel, D. H., & Wiesel, T. N. (1959). Receptive fields of single neurones in the cat's striate cortex. *Journal of Physiology*, *148*(3), 574–591.

Hubel, D. H., & Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *Journal of Physiology*, *160*(1), 106–154.

Hubel, D. H., & Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology*, *195*(1), 215–243.

Hutter, F., Hoos, H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. *Proceedings of the Fifth International Conference on Learning and Intelligent Optimization (LION'11)*.

Hutter, F., Kotthoff, L., & Vanschoren, J. (Eds.) (2019). *Automated Machine Learning: Methods, Systems, Challenges*. Springer.

Ioffe, S. (2017). Batch renormalization: towards reducing minibatch dependence in batch-normalized models. *Advances in Neural Information Processing Systems* (pp. 1945–1953).

Ioffe, S., & Szegedy, C. (2015). Batch normalization: accelerating deep network training by reducing internal covariate shift. *ArXiv:1502.03167*.

Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., & Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. *ArXiv:1803.05407*.

Jacot, A., Gabriel, F., & Hongler, C. (2018). Neural tangent kernel: convergence and generalization in neural networks. *Advances in Neural Information Processing Systems*.

Jaeger, H. (2002). *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network"approach*. GMD-Forschungszentrum Informationstechnik Bonn.

Jamieson, K., & Talwalkar, A. (2016). Non-stochastic best arm identification and hyperparameter optimization. *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*.

Jenatton, R., Archambeau, C., González, J., & Seeger, M. (2017). Bayesian optimization with tree-structured dependencies. *Proceedings of the 34th International Conference on Machine Learning (ICML'17)*.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., ⋯et al. (2018). Highly scalable deep learning training system with mixed-precision: training ImageNet in four minutes. *ArXiv:1807.11205*.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., ⋯Darrell, T. (2014). Caffe: convolutional architecture for fast feature embedding. *Proceedings of the 22nd ACM International Conference on Multimedia* (pp. 675–678).

Joshi, M., Chen, D., Liu, Y., Weld, D. S., Zettlemoyer, L., & Levy, O. (2020). SpanBERT: improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics*, *8*, 64–77.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., ⋯et al. (2017). In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (pp. 1–12).

Kalchbrenner, N., Grefenstette, E., & Blunsom, P. (2014). A convolutional neural network for modelling sentences. *ArXiv:1404.2188*.

Kalman, B. L., & Kwasny, S. C. (1992). Why tanh: choosing a sigmoidal function. *Proceedings of the International Joint Conference on Neural Networks (IJCNN)* (pp. 578–581).

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ⋯Amodei, D. (2020). Scaling laws for neural language models. *ArXiv:2001.08361*.

Karnin, Z., Koren, T., & Somekh, O. (2013). Almost optimal exploration in multi-armed bandits. *Proceedings of the 30th International Conference on Machine Learning (ICML'13)*.

Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of GANs for improved quality, stability, and variation. *ArXiv:1710.10196*.

Kim, J., El-Khamy, M., & Lee, J. (2017). Residual LSTM: design of a deep recurrent architecture for distant speech recognition. *ArXiv:1701.03360*.

Kim, Y. (2014). Convolutional neural networks for sentence classification. *ArXiv:1408.5882*.

Kimeldorf, G. S., & Wahba, G. (1971). Some results on Tchebycheffian spline functions. *J. Math. Anal. Appl.*, *33*, 82–95.

Kingma, D. P., & Ba, J. (2014). Adam: a method for stochastic optimization. *ArXiv:1412.6980*.

Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *ArXiv:1609.02907*.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large language models are zero-shot reasoners. *arxiv.org/abs/2205.11916*.

Koller, D., & Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

Kolmogorov, A. (1933). Sulla determinazione empirica di una legge di distribuzione. *Inst. Ital. Attuari, Giorn.*, *4*, 83–91.

Kolter, Z. (2008). Linear algebra review and reference. *Available online: http://cs229.stanford.edu/section/cs229-linalg.pdf*.

Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, pp. 30–37.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems* (pp. 1097–1105).

Kung, S. Y. (1988). VLSI Array Processors. *Prentice Hall*.

Kuzovkin, I., Vicente, R., Petton, M., Lachaux, J.-P., Baciu, M., Kahane, P., ⋯Aru, J. (2018). Activations of deep convolutional neural networks are aligned with gamma band activity of human visual cortex. *Communications Biology*, *1*(1), 1–12.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., & Soricut, R. (2019). ALBERT: a lite BERT for self-supervised learning of language representations. *ArXiv:1909.11942*.

Lavin, A., & Gray, S. (2016). Fast algorithms for convolutional neural networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4013–4021).

Le, Q. V. (2013). Building high-level features using large scale unsupervised learning. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing* (pp. 8595–8598).

LeCun, Y., Bengio, Y., & et al. (1995). Convolutional networks for images, speech, and time series. *The Handbook of Brain Theory and Neural Networks* (p. 3361). MIT Press.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, *1*(4), 541–551.

LeCun, Y., Bottou, L., Orr, G., & Muller, K.-R. (1998). Efficient backprop. *Neural Networks: Tricks of the Trade*. Springer.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

LeCun, Y., Jackel, L., Bottou, L., Brunot, A., Cortes, C., Denker, J., ⋯et al. (1995). Comparison of learning algorithms for handwritten digit recognition. *International Conference on Artificial Neural Networks* (pp. 53–60).

Legendre, A. M. (1805). *Mémoire sur les Opérations Trigonométriques: dont les Résultats Dépendent de la Figure de la Terre*. F. Didot.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., ⋯Zettlemoyer, L. (2019). BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *ArXiv:1910.13461*.

Lewkowycz, A., Andreassen, A., Dohan, D., Dyer, E., Michalewski, H., Ramasesh, V., ⋯et al. (2022). Solving quantitative reasoning problems with language models. *ArXiv:2206.14858*.

Li, L., Jamieson, K., Rostamizadeh, A., Gonina, K., Hardt, M., Recht, B., & Talwalkar, A. (2018). Massively parallel hyperparameter tuning. *ArXiv:1810.05934*.

Li, M. (2017). *Scaling Distributed Machine Learning with System and Algorithm Co-design* (Doctoral dissertation). PhD Thesis, CMU.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., ⋯Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. *11th Symposium on Operating Systems Design and Implementation (OSDI 14)* (pp. 583–598).

Li, M., Zhang, T., Chen, Y., & Smola, A. J. (2014). Efficient mini-batch training for stochastic optimization. *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 661–670).

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J., & Stoica, I. (2018). Tune: a research platform for distributed model selection and training. *ArXiv:1807.05118*.

Lin, M., Chen, Q., & Yan, S. (2013). Network in network. *ArXiv:1312.4400*.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal loss for dense object detection. *Proceedings of the IEEE International Conference on Computer Vision* (pp. 2980–2988).

Lin, Y., Lv, F., Zhu, S., Yang, M., Cour, T., Yu, K., ⋯others. (2010). ImageNet classification: fast descriptor coding and large-scale SVM training. *Large Scale Visual Recognition Challenge*.

Lin, Z., Feng, M., Santos, C. N. d., Yu, M., Xiang, B., Zhou, B., & Bengio, Y. (2017). A structured self-attentive sentence embedding. *ArXiv:1703.03130*.

Lipton, Z. C., Berkowitz, J., & Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. *ArXiv:1506.00019*.

Lipton, Z. C., Kale, D. C., Elkan, C., & Wetzel, R. (2016). Learning to diagnose with LSTM recurrent neural networks. *International Conference on Learning Representations (ICLR)*.

Lipton, Z. C., & Steinhardt, J. (2018). Troubling trends in machine learning scholarship. *Communications of the ACM*, *17*, 45–77.

Liu, D. C., & Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, *45*(1), 503–528.

Liu, H., Simonyan, K., & Yang, Y. (2018). DARTS: differentiable architecture search. *ArXiv:1806.09055*.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). SSD: single shot multibox detector. *European Conference on Computer Vision* (pp. 21–37).

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ⋯Stoyanov, V. (2019). RoBERTa: a robustly optimized BERT pretraining approach. *ArXiv:1907.11692*.

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., ⋯Guo, B. (2021). Swin transformer: hierarchical vision transformer using shifted windows. *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 10012–10022).

Liu, Z., Mao, H., Wu, C.-Y., Feichtenhofer, C., Darrell, T., & Xie, S. (2022). A convNet for the 2020s. *ArXiv:2201.03545*.

Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 3431–3440).

Loshchilov, I., & Hutter, F. (2016). SGDR: stochastic gradient descent with warm restarts. *ArXiv:1608.03983*.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, *60*(2), 91–110.

Luo, P., Wang, X., Shao, W., & Peng, Z. (2018). Towards understanding regularization in batch normalization. *ArXiv:1809.00846*.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Volume 1* (pp. 142–150).

Mack, Y.-P., & Silverman, B. W. (1982). Weak and strong uniform consistency of kernel regression estimates. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, *61*(3), 405–415.

MacKay, D. J. (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press.

Maclaurin, D., Duvenaud, D., & Adams, R. (2015). Gradient-based hyperparameter optimization through reversible learning. *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*.

Mangasarian, O. L. (1965). Linear and nonlinear separation of patterns by linear programming. *Oper. Res.*, *13*, 444-452.

Mangram, M. E. (2013). A simplified perspective of the Markowitz portfolio theory. *Global Journal of Business Research*, *7*(1), 59–70.

Matthews, A. G. d. G., Rowland, M., Hron, J., Turner, R. E., & Ghahramani, Z. (2018). Gaussian process behaviour in wide deep neural networks. *ArXiv:1804.11271*.

McCann, B., Bradbury, J., Xiong, C., & Socher, R. (2017). Learned in translation: Contextualized word vectors. *Advances in Neural Information Processing Systems* (pp. 6294–6305).

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*(4), 115–133.

McMahan, H. B., Holt, G., Sculley, D., Young, M., Ebner, D., Grady, J., ⋯et al. (2013). Ad click prediction: a view from the trenches. *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 1222–1230).

Mead, C. (1980). Introduction to VLSI systems. *IEE Proceedings I-Solid-State and Electron Devices*, *128*(1), 18.

Merity, S., Xiong, C., Bradbury, J., & Socher, R. (2016). Pointer sentinel mixture models. *ArXiv:1609.07843*.

Micchelli, C. A. (1984). Interpolation of scattered data: distance matrices and conditionally positive definite functions. *Approximation Theory and Spline Functions* (pp. 143–145). Springer.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *ArXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems* (pp. 3111–3119).

Miller, G. A. (1995). WordNet: a lexical database for English. *Communications of the ACM*, *38*(11), 39–41.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., ⋯Dean, J. (2017). Device placement optimization with reinforcement learning. *Proceedings of the 34th International Conference on Machine Learning* (pp. 2430–2439).

Mnih, V., Heess, N., Graves, A., & others. (2014). Recurrent models of visual attention. *Advances in Neural Information Processing Systems* (pp. 2204–2212).

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with deep reinforcement learning. *ArXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ⋯et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Moon, T., Smola, A., Chang, Y., & Zheng, Z. (2010). Intervalrank: isotonic regression with listwise and pairwise constraints. *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining* (pp. 151–160).

Morey, R. D., Hoekstra, R., Rouder, J. N., Lee, M. D., & Wagenmakers, E.-J. (2016). The fallacy of placing confidence in confidence intervals. *Psychonomic Bulletin & Review*, *23*(1), 103–123.

Morozov, V. A. (1984). *Methods for Solving Incorrectly Posed Problems*. Springer.

Nadaraya, E. A. (1964). On estimating regression. *Theory of Probability & its Applications*, *9*(1), 141–142.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. *ICML*.

Nakkiran, P., Kaplun, G., Bansal, Y., Yang, T., Barak, B., & Sutskever, I. (2021). Deep double descent: where bigger models and more data hurt. *Journal of Statistical Mechanics: Theory and Experiment*, *2021*(12), 124003.

Naor, M., & Reingold, O. (1999). On the construction of pseudorandom permutations: Luby–Rackoff revisited. *Journal of Cryptology*, *12*(1), 29–66.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Springer.

Nesterov, Y. (2018). *Lectures on Convex Optimization*. Springer.

Nesterov, Y., & Vial, J.-P. (2000). Confidence level solutions for stochastic programming. *Automatica*, *44*(6), 1559–1568.

Neyman, J. (1937). Outline of a theory of statistical estimation based on the classical theory of probability. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, *236*(767), 333–380.

Norelli, A., Fumero, M., Maiorca, V., Moschella, L., Rodolà, E., & Locatello, F. (2022). ASIF: coupled data turns unimodal models to multimodal without training. *ArXiv:2210.01738*.

Novak, R., Xiao, L., Lee, J., Bahri, Y., Yang, G., Hron, J., ⋯Sohl-Dickstein, J. (2018). Bayesian deep convolutional networks with many channels are Gaussian processes. *ArXiv:1810.05148*.

Novikoff, A. B. J. (1962). On convergence proofs on perceptrons. *Proceedings of the Symposium on the Mathematical Theory of Automata* (pp. 615–622).

Olshausen, B. A., & Field, D. J. (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, *381*(6583), 607–609.

Ong, C. S., Smola, A., & Williamson, R. (2005). Learning the kernel with hyperkernels. *Journal of Machine Learning Research*, *6*, 1043–1071.

OpenAI. (2023). GPT-4 Technical Report. *ArXiv:2303.08774*.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., ⋯ et al. (2022). Training language models to follow instructions with human feedback. *ArXiv:2203.02155*.

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). BLEU: a method for automatic evaluation of machine translation. *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics* (pp. 311–318).

Parikh, A. P., Täckström, O., Das, D., & Uszkoreit, J. (2016). A decomposable attention model for natural language inference. *ArXiv:1606.01933*.

Park, T., Liu, M.-Y., Wang, T.-C., & Zhu, J.-Y. (2019). Semantic image synthesis with spatially-adaptive normalization. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2337–2346).

Parzen, E. (1957). On consistent estimates of the spectrum of a stationary time series. *Annals of Mathematical Statistics*, *28*, 329–348.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ⋯et al. (2019). PyTorch: an imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, *32*, 8026–8037.

Paulus, R., Xiong, C., & Socher, R. (2017). A deep reinforced model for abstractive summarization. *ArXiv:1705.04304*.

Penedo, G., Malartic, Q., Hesslow, D., Cojocaru, R., Cappelli, A., Alobeidli, H., ⋯Launay, J. (2023). The RefinedWeb dataset for Falcon LLM: outperforming curated corpora with web data, and web data only. *ArXiv:2306.01116*.

Pennington, J., Schoenholz, S., & Ganguli, S. (2017). Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. *Advances in Neural Information Processing Systems* (pp. 4785–4795).

Pennington, J., Socher, R., & Manning, C. (2014). GloVe: global vectors for word representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 1532–1543).

Peters, J., Janzing, D., & Schölkopf, B. (2017). *Elements of Causal Inference: Foundations and Learning Algorithms*. MIT Press.

Peters, M., Ammar, W., Bhagavatula, C., & Power, R. (2017). Semi-supervised sequence tagging with bidirectional language models. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Volume 1* (pp. 1756–1765).

Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1* (pp. 2227–2237).

Petersen, K. B., & Pedersen, M. S. (2008). *The Matrix Cookbook*. Technical University of Denmark.

Pleiss, G., Chen, D., Huang, G., Li, T., Van Der Maaten, L., & Weinberger, K. Q. (2017). Memory-efficient implementation of densenets. *ArXiv:1707.06990*.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, *4*(5), 1–17.

Prakash, A., Hasan, S. A., Lee, K., Datla, V., Qadir, A., Liu, J., & Farri, O. (2016). Neural paraphrase generation with stacked residual LSTM networks. *ArXiv:1610.03098*.

Qin, C., Zhang, A., Zhang, Z., Chen, J., Yasunaga, M., & Yang, D. (2023). Is ChatGPT a general-purpose natural language processing task solver? *ArXiv:2302.06476*.

Quadrana, M., Cremonesi, P., & Jannach, D. (2018). Sequence-aware recommender systems. *ACM Computing Surveys*, *51*(4), 66.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Elsevier.

Rabiner, L., & Juang, B.-H. (1993). *Fundamentals of Speech Recognition*. Prentice-Hall.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., ⋯et al. (2021). Learning transferable visual models from natural language supervision. *International Conference on Machine Learning* (pp. 8748–8763).

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *ArXiv:1511.06434*.

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training. *OpenAI*.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, *1*(8), 9.

Radosavovic, I., Johnson, J., Xie, S., Lo, W.-Y., & Dollár, P. (2019). On network design spaces for visual recognition. *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 1882–1890).

Radosavovic, I., Kosaraju, R. P., Girshick, R., He, K., & Dollár, P. (2020). Designing network design spaces. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 10428–10436).

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., ⋯et al. (2021). Scaling language models: methods, analysis & insights from training gopher. *ArXiv:2112.11446*.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., ⋯Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, *21*, 1–67.

Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016). SQuAD: 100,000+ questions for machine comprehension of text. *ArXiv:1606.05250*.

Ramachandran, P., Parmar, N., Vaswani, A., Bello, I., Levskaya, A., & Shlens, J. (2019). Stand-alone self-attention in vision models. *Advances in Neural Information Processing Systems*, *32*.

Ramachandran, P., Zoph, B., & Le, Q. V. (2017). Searching for activation functions. *ArXiv:1710.05941*.

Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., & Chen, M. (2022). Hierarchical text-conditional image generation with clip latents. *ArXiv:2204.06125*.

Ramón y Cajal, Santiago, & Azoulay, L. (1894). *Les Nouvelles Idées sur la Structure du Système Nerveux chez l'Homme et chez les Vertébrés*. Paris, C. Reinwald & Cie.

Ranzato, M.-A., Boureau, Y.-L., Chopra, S., & LeCun, Y. (2007). A unified energy-based framework for unsupervised learning. *Artificial Intelligence and Statistics* (pp. 371–379).

Rasmussen, C. E., & Williams, C. K. (2006). *Gaussian Processes for Machine Learning*. MIT Press.

Reddi, S. J., Kale, S., & Kumar, S. (2019). On the convergence of Adam and beyond. *ArXiv:1904.09237*.

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: unified, real-time object detection. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 779–788).

Redmon, J., & Farhadi, A. (2018). YOLOv3: an incremental improvement. *ArXiv:1804.02767*.

Reed, S., & De Freitas, N. (2015). Neural programmer-interpreters. *ArXiv:1511.06279*.

Reed, S., Zolna, K., Parisotto, E., Colmenarejo, S. G., Novikov, A., Barth-Maron, G., ⋯ et al. (2022). A generalist agent. *ArXiv:2205.06175*.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: towards real-time object detection with region proposal networks. *Advances in Neural Information Processing Systems* (pp. 91–99).

Rendle, S. (2010). Factorization machines. *2010 IEEE International Conference on Data Mining* (pp. 995–1000).

Rendle, S., Freudenthaler, C., Gantner, Z., & Schmidt-Thieme, L. (2009). BPR: Bayesian personalized ranking from implicit feedback. *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence* (pp. 452–461).

Revels, J., Lubin, M., & Papamarkou, T. (2016). Forward-mode automatic differentiation in Julia. *ArXiv:1607.07892*.

Riesenhuber, M., & Poggio, T. (1999). Hierarchical models of object recognition in cortex. *Nature Neuroscience*, *2*(11), 1019–1025.

Rockafellar, R. T. (1970). *Convex Analysis*. Princeton University Press.

Rolnick, D., Veit, A., Belongie, S., & Shavit, N. (2017). Deep learning is robust to massive label noise. *ArXiv:1705.10694*.

Rudin, W. (1973). *Functional Analysis*. McGraw-Hill.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive Modeling*, *5*(3), 1.

Russakovsky, O., Deng, J., Huang, Z., Berg, A. C., & Fei-Fei, L. (2013). Detecting avocados to zucchinis: what have we done, and where are we going? *International Conference on Computer Vision (ICCV)*.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., ⋯et al. (2015). ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, *115*(3), 211–252.

Russell, S. J., & Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Pearson Education Limited.

Saharia, C., Chan, W., Saxena, S., Li, L., Whang, J., Denton, E., ⋯et al. (2022). Photorealistic text-to-image diffusion models with deep language understanding. *ArXiv:2205.11487*.

Salinas, D., Seeger, M., Klein, A., Perrone, V., Wistuba, M., & Archambeau, C. (2022). Syne Tune: a library for large scale hyperparameter tuning and reproducible research. *First Conference on Automated Machine Learning*.

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *ArXiv:1910.01108*.

Sanh, V., Webson, A., Raffel, C., Bach, S. H., Sutawika, L., Alyafeai, Z., ⋯et al. (2021). Multitask prompted training enables zero-shot task generalization. *ArXiv:2110.08207*.

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? *Advances in Neural Information Processing Systems* (pp. 2483–2493).

Sarwar, B. M., Karypis, G., Konstan, J. A., & Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. *Proceedings of 10th International Conference on World Wide Web* (pp. 285–295).

Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ilić, S., Hesslow, D., ⋯et al. (2022). BLOOM: a 176B-parameter open-access multilingual language model. *ArXiv:2211.05100*.

Schein, A. I., Popescul, A., Ungar, L. H., & Pennock, D. M. (2002). Methods and metrics for cold-start recommendations. *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 253–260).

Schuhmann, C., Beaumont, R., Vencu, R., Gordon, C., Wightman, R., Cherti, M., ⋯et al. (2022). LAION-5B: an open large-scale dataset for training next generation image-text models. *ArXiv:2210.08402*.

Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, *45*(11), 2673–2681.

Schölkopf, B., Herbrich, R., & Smola, A. J. (2001). Helmbold, D. P., & Williamson, B. (Eds.). A generalized representer theorem. *Proceedings of the Annual Conference on Computational Learning Theory* (pp. 416–426). Springer-Verlag.

Schölkopf, B., Burges, C., & Vapnik, V. (1996). Incorporating invariances in support vector learning machines. *International Conference on Artificial Neural Networks* (pp. 47–52).

Schölkopf, B., & Smola, A. J. (2002). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press.

Sedhain, S., Menon, A. K., Sanner, S., & Xie, L. (2015). Autorec: autoencoders meet collaborative filtering. *Proceedings of the 24th International Conference on World Wide Web* (pp. 111–112).

Sennrich, R., Haddow, B., & Birch, A. (2015). Neural machine translation of rare words with subword units. *ArXiv:1508.07909*.

Sergeev, A., & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in TensorFlow. *ArXiv:1802.05799*.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, *27*(3), 379–423.

Shao, H., Yao, S., Sun, D., Zhang, A., Liu, S., Liu, D., ⋯Abdelzaher, T. (2020). ControlVAE: controllable variational autoencoder. *Proceedings of the 37th International Conference on Machine Learning*.

Shaw, P., Uszkoreit, J., & Vaswani, A. (2018). Self-attention with relative position representations. *ArXiv:1803.02155*.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019). Megatron-LM: training multi-billion parameter language models using model parallelism. *ArXiv:1909.08053*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ⋯et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484.

Silverman, B. W. (1986). *Density Estimation for Statistical and Data Analysis*. Chapman and Hall.

Simard, P. Y., LeCun, Y. A., Denker, J. S., & Victorri, B. (1998). Transformation invariance in pattern recognition −tangent distance and tangent propagation. *Neural Networks: Tricks of the Trade* (pp. 239−274). Springer.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *ArXiv:1409.1556*.

Sindhwani, V., Sainath, T. N., & Kumar, S. (2015). Structured transforms for small-footprint deep learning. *ArXiv:1510.01722*.

Sivic, J., & Zisserman, A. (2003). Video Google: a text retrieval approach to object matching in videos. *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1470−1470).

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., ⋯et al. (2022). Using DeepSpeed and Megatron to train Megatron-Turing NLG 530B, a large-scale generative language model. *ArXiv:2201.11990*.

Smola, A., & Narayanamurthy, S. (2010). An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, *3*(1-2), 703−710.

Snoek, J., Larochelle, H., & Adams, R. (2012). Practical Bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems 25* (pp. 2951−2959).

Sohl-Dickstein, J., Weiss, E., Maheswaranathan, N., & Ganguli, S. (2015). Deep unsupervised learning using nonequilibrium thermodynamics. *International Conference on Machine Learning* (pp. 2256−2265).

Speelpenning, B. (1980). *Compiling fast partial derivatives of functions given by algorithms* (Doctoral dissertation). University of Illinois at Urbana-Champaign.

Srivastava, A., Rastogi, A., Rao, A., Shoeb, A. A. M., Abid, A., Fisch, A., ⋯et al. (2022). Beyond the imitation game: quantifying and extrapolating the capabilities of language models. *ArXiv:2206.04615*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(1), 1929−1958.

Srivastava, R. K., Greff, K., & Schmidhuber, J. (2015). Highway networks. *ArXiv:1505.00387*.

Strang, G. (1993). *Introduction to Linear Algebra*. Wellesley−Cambridge Press.

Su, X., & Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in Artificial Intelligence*, *2009*.

Sukhbaatar, S., Weston, J., & Fergus, R. (2015). End-to-end memory networks. *Advances in Neural Information Processing Systems* (pp. 2440–2448).

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *International Conference on Machine Learning* (pp. 1139–1147).

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems* (pp. 3104–3112).

Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017). Inception-v4, Inception-ResNet and the impact of residual connections on learning. *31st AAAI Conference on Artificial Intelligence*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ⋯Rabinovich, A. (2015). Going deeper with convolutions. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1–9).

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the Inception architecture for computer vision. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2818–2826).

Tallec, C., & Ollivier, Y. (2017). Unbiasing truncated backpropagation through time. *ArXiv:1705.08209*.

Tan, M., & Le, Q. (2019). EfficientNet: rethinking model scaling for convolutional neural networks. *International Conference on Machine Learning* (pp. 6105–6114).

Tang, J., & Wang, K. (2018). Personalized top-n sequential recommendation via convolutional sequence embedding. *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* (pp. 565–573).

Taskar, B., Guestrin, C., & Koller, D. (2004). Max-margin Markov networks. *Advances in Neural Information Processing Systems*, *16*, 25.

Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2020). Efficient transformers: a survey. *ArXiv:2009.06732*.

Taylor, R., Kardas, M., Cucurull, G., Scialom, T., Hartshorn, A., Saravia, E., ⋯Stojnic, R. (2022). Galactica: a large language model for science. *ArXiv:2211.09085*.

Teye, M., Azizpour, H., & Smith, K. (2018). Bayesian uncertainty estimation for batch normalized deep networks. *ArXiv:1802.06455*.

Thomee, B., Shamma, D. A., Friedland, G., Elizalde, B., Ni, K., Poland, D., ⋯Li, L.-J. (2016). Yfcc100m: the new data in multimedia research. *Communications of the ACM*, *59*(2), 64–73.

Tieleman, T., & Hinton, G. (2012). Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning, Lecture 6.5-rmsprop*.

Tikhonov, A. N., & Arsenin, V. Y. (1977). *Solutions of Ill-Posed Problems*. W.H. Winston.

Tolstikhin, I. O., Houlsby, N., Kolesnikov, A., Beyer, L., Zhai, X., Unterthiner, T., ⋯et al. (2021). MLP-mixer: an all-MLP architecture for vision. *Advances in Neural Information Processing Systems*, *34*.

Torralba, A., Fergus, R., & Freeman, W. T. (2008). 80 million tiny images: a large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *30*(11), 1958–1970.

Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., & Jégou, H. (2021). Training data-efficient image transformers & distillation through attention. *International Conference on Machine Learning* (pp. 10347–10357).

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., ⋯et al. (2023a). LLaMA: open and efficient foundation language models. *ArXiv:2302.13971*.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., ⋯et al. (2023b). LLaMA 2: open foundation and fine-tuned chat models. *ArXiv:2307.09288*.

Tsoumakas, G., & Katakis, I. (2007). Multi-label classification: an overview. *International Journal of Data Warehousing and Mining*, *3*(3), 1–13.

Turing, A. (1950). Computing machinery and intelligence. *Mind*, *59*(236), 433.

Töscher, A., Jahrer, M., & Bell, R. M. (2009). *The bigchaos solution to the Netflix grand prize*.

Uijlings, J. R., Van De Sande, K. E., Gevers, T., & Smeulders, A. W. (2013). Selective search for object recognition. *International Journal of Computer Vision*, *104*(2), 154–171.

Vapnik, V. (1995). *The Nature of Statistical Learning Theory*. New York: Springer.

Vapnik, V. (1998). *Statistical Learning Theory*. New York: John Wiley and Sons.

Vapnik, V., & Chervonenkis, A. (1964). A note on one class of perceptrons. *Automation and Remote Control*, *25*.

Vapnik, V., & Chervonenkis, A. (1968). Uniform convergence of frequencies of occurence of events to their probabilities. *Dokl. Akad. Nauk SSSR*, *181*, 915-918.

Vapnik, V., & Chervonenkis, A. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory Probab. Appl.*, *16*(2), 264-281.

Vapnik, V., & Chervonenkis, A. (1981). The necessary and sufficient conditions for the uniform convergence of averages to their expected values. *Teoriya Veroyatnostei i Ee Primeneniya*, *26*(3), 543-564.

Vapnik, V., & Chervonenkis, A. (1991). The necessary and sufficient conditions for consistency in the empirical risk minimization method. *Pattern Recognition and Image Analysis*, *1*(3), 283-305.

Vapnik, V. N., & Chervonenkis, A. Y. (1974). Ordered risk minimization. *Automation and Remote Control*, *35*, 1226–1235, 1403–1412.

Vapnik, V. (1992). Principles of risk minimization for learning theory. *Advances in Neural Information Processing Systems* (pp. 831–838).

Vapnik, V., Levin, E., & Le Cun, Y. (1994). Measuring the VC-dimension of a learning machine. *Neural Computation*, *6*(5), 851–876.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ⋯Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems* (pp. 5998–6008).

Wahba, G. (1990). *Spline Models for Observational Data*. SIAM.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. J. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *37*(3), 328–339.

Wang, H., Zhang, A., Zheng, S., Shi, X., Li, M., & Wang, Z. (2022). Removing batch normalization boosts adversarial training. *International Conference on Machine Learning* (pp. 23433–23445).

Wang, L., Li, M., Liberty, E., & Smola, A. J. (2018). Optimal message scheduling for aggregation. *Networks*, *2*(3), 2–3.

Wang, Q., Li, B., Xiao, T., Zhu, J., Li, C., Wong, D. F., & Chao, L. S. (2019). Learning deep transformer models for machine translation. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (pp. 1810–1822).

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., & Zhou, D. (2023). Self-consistency improves chain of thought reasoning in language models. *International Conference on Learning Representations*.

Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., & Owens, J. D. (2016). Gunrock: a high-performance graph processing library on the GPU. *ACM SIGPLAN Notices* (p. 11).

Warstadt, A., Singh, A., & Bowman, S. R. (2019). Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, *7*, 625–641.

Wasserman, L. (2013). *All of Statistics: A Concise Course in Statistical Inference*. Springer.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3–4), 279–292.

Watson, G. S. (1964). Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics, Series A*, pp. 359–372.

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., ⋯Le, Q. V. (2021). Finetuned language models are zero-shot learners. *ArXiv:2109.01652*.

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., ⋯et al. (2022). Emergent abilities of large language models. *ArXiv:2206.07682*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., & Zhou, D. (2022). Chain of thought prompting elicits reasoning in large language models. *ArXiv:2201.11903*.

Welling, M., & Teh, Y. W. (2011). Bayesian learning via stochastic gradient Langevin dynamics. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (pp. 681–688).

Wengert, R. E. (1964). A simple automatic derivative evaluation program. *Communications of the ACM*, *7*(8), 463–464.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560.

Wigner, E. P. (1958). On the distribution of the roots of certain symmetric matrices. *Ann. Math.* (pp. 325–327).

Wilson, A. G., & Izmailov, P. (2020). Bayesian deep learning and a probabilistic perspective of generalization. *Advances in Neural Information Processing Systems*, *33*, 4697–4708.

Wistuba, M., Rawat, A., & Pedapati, T. (2019). A survey on neural architecture search. *ArXiv:1905.01392 [cs.LG]*.

Wistuba, M., Schilling, N., & Schmidt-Thieme, L. (2018). Scalable Gaussian process-based transfer surrogates for hyperparameter optimization. *Machine Learning*, *108*, 43–78.

Wolpert, D. H., & Macready, W. G. (1995). *No free lunch theorems for search*. Technical Report SFI-TR-95-02-010, Santa Fe Institute.

Wood, F., Gasthaus, J., Archambeau, C., James, L., & Teh, Y. W. (2011). The sequence memoizer. *Communications of the ACM*, *54*(2), 91–98.

Wu, B., Wan, A., Yue, X., Jin, P., Zhao, S., Golmant, N., ⋯Keutzer, K. (2018). Shift: a zero flop, zero parameter alternative to spatial convolutions. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 9127–9135).

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., ⋯et al. (2016). Google's neural machine translation system: bridging the gap between human and machine translation. *ArXiv:1609.08144*.

Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *ArXiv:1708.07747*.

Xiao, L., Bahri, Y., Sohl-Dickstein, J., Schoenholz, S., & Pennington, J. (2018). Dynamical isometry and a mean field theory of CNNs: how to train 10,000-layer vanilla convolutional neural networks. *International Conference on Machine Learning* (pp. 5393–5402).

Xie, S., Girshick, R., Dollár, P., Tu, Z., & He, K. (2017). Aggregated residual transformations for deep neural networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1492–1500).

Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., ⋯Liu, T. (2020). On layer normalization in the transformer architecture. *International Conference on Machine Learning* (pp. 10524–10533).

Xiong, W., Wu, L., Alleva, F., Droppo, J., Huang, X., & Stolcke, A. (2018). The Microsoft 2017 conversational speech recognition system. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5934–5938).

Yamaguchi, K., Sakamoto, K., Akabane, T., & Fujimoto, Y. (1990). A neural network for speaker-independent isolated word recognition. *First International Conference on Spoken Language Processing*.

Yang, Z., Hu, Z., Deng, Y., Dyer, C., & Smola, A. (2016). Neural machine translation with recurrent attention modeling. *ArXiv:1607.05108*.

Yang, Z., Moczulski, M., Denil, M., De Freitas, N., Smola, A., Song, L., & Wang, Z. (2015). Deep fried convnets. *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1476–1483).

Ye, M., Yin, P., Lee, W.-C., & Lee, D.-L. (2011). Exploiting geographical influence for collaborative point-of-interest recommendation. *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 325–334).

You, Y., Gitman, I., & Ginsburg, B. (2017). Large batch training of convolutional networks. *ArXiv:1708.03888*.

Yu, J., Xu, Y., Koh, J. Y., Luong, T., Baid, G., Wang, Z., ⋯Wu, Y. (2022). Scaling autoregressive models for content-rich text-to-image generation. *ArXiv:2206.10789*.

Zaheer, M., Reddi, S., Sachan, D., Kale, S., & Kumar, S. (2018). Adaptive methods for nonconvex optimization. *Advances in Neural Information Processing Systems* (pp. 9793–9803).

Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *ArXiv:1212.5701*.

Zeiler, M. D., & Fergus, R. (2013). Stochastic pooling for regularization of deep convolutional neural networks. *ArXiv:1301.3557*.

Zhang, A., Tay, Y., Zhang, S., Chan, A., Luu, A. T., Hui, S. C., & Fu, J. (2021). Beyond fully-connected layers with quaternions: parameterization of hypercomplex multiplications with 1/n parameters. *International Conference on Learning Representations*.

Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2021). Understanding deep

learning (still) requires rethinking generalization. *Communications of the ACM*, *64*(3), 107–115.

Zhang, S., Yao, L., Sun, A., & Tay, Y. (2019). Deep learning based recommender system: a survey and new perspectives. *ACM Computing Surveys*, *52*(1), 5.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., ⋯et al. (2022). OPT: open pre-trained transformer language models. *ArXiv:2205.01068*.

Zhang, W., Tanida, J., Itoh, K., & Ichioka, Y. (1988). Shift-invariant pattern recognition neural network and its optical architecture. *Proceedings of Annual Conference of the Japan Society of Applied Physics*.

Zhang, Y., Sun, P., Jiang, Y., Yu, D., Yuan, Z., Luo, P., ⋯Wang, X. (2021). ByteTrack: multi-object tracking by associating every detection box. *ArXiv:2110.06864*.

Zhang, Z., Zhang, A., Li, M., & Smola, A. (2023). Automatic chain of thought prompting in large language models. *International Conference on Learning Representations*.

Zhang, Z., Zhang, A., Li, M., Zhao, H., Karypis, G., & Smola, A. (2023). Multimodal chain-of-thought reasoning in language models. *ArXiv:2302.00923*.

Zhao, Z.-Q., Zheng, P., Xu, S.-t., & Wu, X. (2019). Object detection with deep learning: a review. *IEEE Transactions on Neural Networks and Learning Systems*, *30*(11), 3212–3232.

Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., ⋯Chi, E. (2023). Least-to-most prompting enables complex reasoning in large language models. *International Conference on Learning Representations*.

Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. *Proceedings of the IEEE International Conference on Computer Vision* (pp. 2223–2232).

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., & Fidler, S. (2015). Aligning books and movies: towards story-like visual explanations by watching movies and reading books. *Proceedings of the IEEE International Conference on Computer Vision* (pp. 19–27).

Zoph, B., & Le, Q. V. (2016). Neural architecture search with reinforcement learning. *ArXiv:1611.01578*.