Dive into Deep Learning

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

Contents

Preface

Just a few years ago, there were no legions of deep learning scientists developing intelligent products and services at major companies and startups. When we entered the field, machine learning did not command headlines in daily newspapers. Our parents had no idea what machine learning was, let alone why we might prefer it to a career in medicine or law. Machine learning was a blue skies academic discipline whose industrial significance was limited to a narrow set of real-world applications, including speech recognition and computer vision. Moreover, many of these applications required so much domain knowledge that they were often regarded as entirely separate areas for which machine learning was one small component. At that time, neural networks—the predecessors of the deep learning methods that we focus on in this book—were generally regarded as outmoded.

Yet in just few years, deep learning has taken the world by surprise, driving rapid progress in such diverse fields as computer vision, natural language processing, automatic speech recognition, reinforcement learning, and biomedical informatics. Moreover, the success of deep learning in so many tasks of practical interest has even catalyzed developments in theoretical machine learning and statistics. With these advances in hand, we can now build cars that drive themselves with more autonomy than ever before (though less autonomy than some companies might have you believe), dialogue systems that debug code by asking clarifying questions, and software agents beating the best human players in the world at board games such as Go, a feat once thought to be decades away. Already, these tools exert ever-wider influence on industry and society, changing the way movies are made, diseases are diagnosed, and playing a growing role in basic sciences—from astrophysics, to climate modeling, to weather prediction, to biomedicine.

About This Book

This book represents our attempt to make deep learning approachable, teaching you the *concepts*, the *context*, and the *code*.

One Medium Combining Code, Math, and HTML

For any computing technology to reach its full impact, it must be well understood, well documented, and supported by mature, well-maintained tools. The key ideas should be clearly distilled, minimizing the onboarding time needed to bring new practitioners up to

date. Mature libraries should automate common tasks, and exemplar code should make it easy for practitioners to modify, apply, and extend common applications to suit their needs.

As an example, take dynamic web applications. Despite a large number of companies, such as Amazon, developing successful database-driven web applications in the 1990s, the potential of this technology to aid creative entrepreneurs was realized to a far greater degree only in the past ten years, owing in part to the development of powerful, well-documented frameworks.

Testing the potential of deep learning presents unique challenges because any single application brings together various disciplines. Applying deep learning requires simultaneously understanding (i) the motivations for casting a problem in a particular way; (ii) the mathematical form of a given model; (iii) the optimization algorithms for fitting the models to data; (iv) the statistical principles that tell us when we should expect our models to generalize to unseen data and practical methods for certifying that they have, in fact, generalized; and (v) the engineering techniques required to train models efficiently, navigating the pitfalls of numerical computing and getting the most out of available hardware. Teaching the critical thinking skills required to formulate problems, the mathematics to solve them, and the software tools to implement those solutions all in one place presents formidable challenges. Our goal in this book is to present a unified resource to bring would-be practitioners up to speed.

When we started this book project, there were no resources that simultaneously (i) remained up to date; (ii) covered the breadth of modern machine learning practices with sufficient technical depth; and (iii) interleaved exposition of the quality one expects of a textbook with the clean runnable code that one expects of a hands-on tutorial. We found plenty of code examples illustrating how to use a given deep learning framework (e.g., how to do basic numerical computing with matrices in TensorFlow) or for implementing particular techniques (e.g., code snippets for LeNet, AlexNet, ResNet, etc.) scattered across various blog posts and GitHub repositories. However, these examples typically focused on *how* to implement a given approach, but left out the discussion of *why* certain algorithmic decisions are made. While some interactive resources have popped up sporadically to address a particular topic, e.g., the engaging blog posts published on the website Distill[1], or personal blogs, they only covered selected topics in deep learning, and often lacked associated code. On the other hand, while several deep learning textbooks have emerged—e.g., Goodfellow *et al.* (2016), which offers a comprehensive survey on the basics of deep learning—these resources do not marry the descriptions to realizations of the concepts in code, sometimes leaving readers clueless as to how to implement them. Moreover, too many resources are hidden behind the paywalls of commercial course providers.


1

We set out to create a resource that could (i) be freely available for everyone; (ii) offer sufficient technical depth to provide a starting point on the path to actually becoming an applied machine learning scientist; (iii) include runnable code, showing readers *how* to solve problems in practice; (iv) allow for rapid updates, both by us and also by the community at large; and (v) be complemented by a forum[2] for interactive discussion of technical details and to answer questions.


2

These goals were often in conflict. Equations, theorems, and citations are best managed and laid out in LaTeX. Code is best described in Python. And webpages are native in HTML and JavaScript. Furthermore, we want the content to be accessible both as executable code, as a physical book, as a downloadable PDF, and on the Internet as a website. No workflows seemed suited to these demands, so we decided to assemble our own (Section B.6). We settled on GitHub to share the source and to facilitate community contributions; Jupyter notebooks for mixing code, equations and text; Sphinx as a rendering engine; and Discourse as a discussion platform. While our system is not perfect, these choices strike a compromise among the competing concerns. We believe that *Dive into Deep Learning* might be the first book published using such an integrated workflow.

Learning by Doing

Many textbooks present concepts in succession, covering each in exhaustive detail. For example, the excellent textbook of Bishop (2006), teaches each topic so thoroughly that getting to the chapter on linear regression requires a nontrivial amount of work. While experts love this book precisely for its thoroughness, for true beginners, this property limits its usefulness as an introductory text.

In this book, we teach most concepts *just in time*. In other words, you will learn concepts at the very moment that they are needed to accomplish some practical end. While we take some time at the outset to teach fundamental preliminaries, like linear algebra and probability, we want you to taste the satisfaction of training your first model before worrying about more esoteric concepts.

Aside from a few preliminary notebooks that provide a crash course in the basic mathematical background, each subsequent chapter both introduces a reasonable number of new concepts and provides several self-contained working examples, using real datasets. This presented an organizational challenge. Some models might logically be grouped together in a single notebook. And some ideas might be best taught by executing several models in succession. By contrast, there is a big advantage to adhering to a policy of *one working example, one notebook*: This makes it as easy as possible for you to start your own research projects by leveraging our code. Just copy a notebook and start modifying it.

Throughout, we interleave the runnable code with background material as needed. In general, we err on the side of making tools available before explaining them fully (often filling in the background later). For instance, we might use *stochastic gradient descent* before explaining why it is useful or offering some intuition for why it works. This helps to give practitioners the necessary ammunition to solve problems quickly, at the expense of requiring the reader to trust us with some curatorial decisions.

This book teaches deep learning concepts from scratch. Sometimes, we delve into fine details about models that would typically be hidden from users by modern deep learning frameworks. This comes up especially in the basic tutorials, where we want you to understand everything that happens in a given layer or optimizer. In these cases, we often present two versions of the example: one where we implement everything from scratch, relying only on NumPy-like functionality and automatic differentiation, and a more prac-

tical example, where we write succinct code using the high-level APIs of deep learning frameworks. After explaining how some component works, we rely on the high-level API in subsequent tutorials.

Content and Structure

The book can be divided into roughly three parts, dealing with preliminaries, deep learning techniques, and advanced topics focused on real systems and applications (Fig. 1).

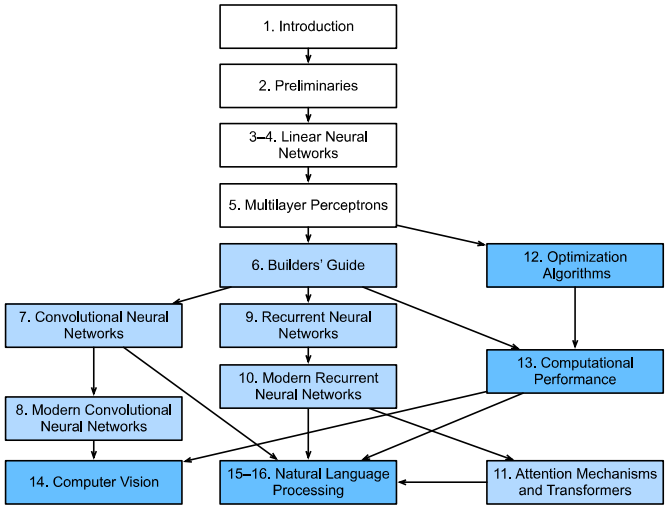


Fig. 1 Book structure.

- **Part 1: Basics and Preliminaries**. Chapter 1 is an introduction to deep learning. Then, in Chapter 2, we quickly bring you up to speed on the prerequisites required for hands-on deep learning, such as how to store and manipulate data, and how to apply various numerical operations based on elementary concepts from linear algebra, calculus, and probability. Chapter 3 and Chapter 5 cover the most fundamental concepts and techniques in deep learning, including regression and classification; linear models; multilayer perceptrons; and overfitting and regularization.

- **Part 2: Modern Deep Learning Techniques**. Chapter 6 describes the key computational components of deep learning systems and lays the groundwork for our subsequent implementations of more complex models. Next, Chapter 7 and Chapter 8 present convolutional neural networks (CNNs), powerful tools that form the backbone of most modern computer vision systems. Similarly, Chapter 9 and Chapter 10 introduce recurrent neural networks (RNNs), models that exploit sequential (e.g., temporal) structure in data and are commonly used for natural language processing and time series prediction. In Chapter 11, we describe a relatively new class of models, based on so-called *attention mechanisms*, that has displaced RNNs as the dominant architecture for most natural language processing tasks. These sections will bring you up to speed on the most powerful and general tools that are widely used by deep learning practitioners.

3 

- **Part 3: Scalability, Efficiency, and Applications** (available online[3]). In Chapter 12, we discuss several common optimization algorithms used to train deep learning models. Next, in Chapter 13, we examine several key factors that influence the computational performance of deep learning code. Then, in Chapter 14, we illustrate major applications of deep learning in computer vision. Finally, in Chapter 15 and Chapter 16, we demonstrate how to pretrain language representation models and apply them to natural language processing tasks.

Code

Most sections of this book feature executable code. We believe that some intuitions are best developed via trial and error, tweaking the code in small ways and observing the results. Ideally, an elegant mathematical theory might tell us precisely how to tweak our code to achieve a desired result. However, deep learning practitioners today must often tread where no solid theory provides guidance. Despite our best attempts, formal explanations for the efficacy of various techniques are still lacking, for a variety of reasons: the mathematics to characterize these models can be so difficult; the explanation likely depends on properties of the data that currently lack clear definitions; and serious inquiry on these topics has only recently kicked into high gear. We are hopeful that as the theory of deep learning progresses, each future edition of this book will provide insights that eclipse those presently available.

To avoid unnecessary repetition, we capture some of our most frequently imported and used functions and classes in the `d2l` package. Throughout, we mark blocks of code (such as functions, classes, or collection of import statements) with `#@save` to indicate that they will be accessed later via the `d2l` package. We offer a detailed overview of these classes and functions in Section B.8. The `d2l` package is lightweight and only requires the following dependencies:

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

d2l = sys.modules[__name__]
```

Most of the code in this book is based on PyTorch, a popular open-source framework that has been enthusiastically embraced by the deep learning research community. All of the code in this book has passed tests under the latest stable version of PyTorch. However, due to the rapid development of deep learning, some code *in the print edition* may not work properly in future versions of PyTorch. We plan to keep the online version up to date. In case you encounter any problems, please consult *Installation* (page xxxiv) to update your code and runtime environment. Below lists dependencies in our PyTorch implementation.

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

Target Audience

This book is for students (undergraduate or graduate), engineers, and researchers, who seek a solid grasp of the practical techniques of deep learning. Because we explain every concept from scratch, no previous background in deep learning or machine learning is required. Fully explaining the methods of deep learning requires some mathematics and programming, but we will only assume that you enter with some basics, including modest amounts of linear algebra, calculus, probability, and Python programming. Just in case you have forgotten anything, the online Appendix[4] provides a refresher on most of the mathematics you will find in this book. Usually, we will prioritize intuition and ideas over mathematical rigor. If you would like to extend these foundations beyond the prerequisites to understand our book, we happily recommend some other terrific resources: *Linear Analysis* by Bollobás (1999) covers linear algebra and functional analysis in great depth. *All of Statistics* (Wasserman, 2013) provides a marvelous introduction to statistics. Joe Blitzstein's books[5] and courses[6] on probability and inference are pedagogical gems. And if you have not used Python before, you may want to peruse this Python tutorial[7].

Notebooks, Website, GitHub, and Forum

All of our notebooks are available for download on the D2L.ai website[8] and on GitHub[9]. Associated with this book, we have launched a discussion forum, located at discuss.d2l.ai [10]. Whenever you have questions on any section of the book, you can find a link to the associated discussion page at the end of each notebook.

Acknowledgments

We are indebted to the hundreds of contributors for both the English and the Chinese drafts. They helped improve the content and offered valuable feedback. This book was originally implemented with MXNet as the primary framework. We thank Anirudh Dagar and Yuan Tang for adapting a majority part of earlier MXNet code into PyTorch and TensorFlow implementations, respectively. Since July 2021, we have redesigned and reimplemented this book in PyTorch, MXNet, and TensorFlow, choosing PyTorch as the primary framework. We thank Anirudh Dagar for adapting a majority part of more recent PyTorch code into JAX implementations. We thank Gaosheng Wu, Liujun Hu, Ge Zhang, and Jiehang Xie from Baidu for adapting a majority part of more recent PyTorch code into PaddlePaddle implementations in the Chinese draft. We thank Shuai Zhang for integrating the LaTeX style from the press into the PDF building.

On GitHub, we thank every contributor of this English draft for making it better for everyone. Their GitHub IDs or names are (in no particular order): 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.

We thank Amazon Web Services, especially Wen-Ming Ye, George Karypis, Swami Sivasubramanian, Peter DeSantis, Adam Selipsky, and Andrew Jassy for their generous support in writing this book. Without the available time, resources, discussions with colleagues, and continuous encouragement, this book would not have happened. During the preparation of the book for publication, Cambridge University Press has offered excellent support. We thank our commissioning editor David Tranah for his help and professionalism.

Summary

Deep learning has revolutionized pattern recognition, introducing technology that now powers a wide range of technologies, in such diverse fields as computer vision, natural language processing, and automatic speech recognition. To successfully apply deep learning, you must understand how to cast a problem, the basic mathematics of modeling, the algorithms for fitting your models to data, and the engineering techniques to implement it all. This book presents a comprehensive resource, including prose, figures, mathematics, and code, all in one place.

Exercises

1. Register an account on the discussion forum of this book discuss.d2l.ai[11].
2. Install Python on your computer.

3. Follow the links at the bottom of the section to the forum, where you will be able to seek out help and discuss the book and find answers to your questions by engaging the authors and broader community.

Discussions[12].

[12] 

Installation

In order to get up and running, we will need an environment for running Python, the Jupyter Notebook, the relevant libraries, and the code needed to run the book itself.

Installing Miniconda

13 

Your simplest option is to install Miniconda[13]. Note that the Python 3.x version is required. You can skip the following steps if your machine already has conda installed.

Visit the Miniconda website and determine the appropriate version for your system based on your Python 3.x version and machine architecture. Suppose that your Python version is 3.9 (our tested version). If you are using macOS, you would download the bash script whose name contains the strings "MacOSX", navigate to the download location, and execute the installation as follows (taking Intel Macs as an example):

```
# The file name is subject to changes
sh Miniconda3-py39_4.12.0-MacOSX-x86_64.sh -b
```

A Linux user would download the file whose name contains the strings "Linux" and execute the following at the download location:

```
# The file name is subject to changes
sh Miniconda3-py39_4.12.0-Linux-x86_64.sh -b
```

14 

A Windows user would download and install Miniconda by following its online instructions [14]. On Windows, you may search for `cmd` to open the Command Prompt (command-line interpreter) for running commands.

Next, initialize the shell so we can run `conda` directly.

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

Then close and reopen your current shell. You should be able to create a new environment as follows:

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

Now we can activate the d2l environment:

```
conda activate d2l
```

Installing the Deep Learning Framework and the d2l Package

Before installing any deep learning framework, please first check whether or not you have proper GPUs on your machine (the GPUs that power the display on a standard laptop are not relevant for our purposes). For example, if your computer has NVIDIA GPUs and has installed CUDA [15], then you are all set. If your machine does not house any GPU, there is no need to worry just yet. Your CPU provides more than enough horsepower to get you through the first few chapters. Just remember that you will want to access GPUs before running larger models.

You can install PyTorch (the specified versions are tested at the time of writing) with either CPU or GPU support as follows:

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

Our next step is to install the d2l package that we developed in order to encapsulate frequently used functions and classes found throughout this book:

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

Downloading and Running the Code

Next, you will want to download the notebooks so that you can run each of the book's code blocks. Simply click on the "Notebooks" tab at the top of any HTML page on the D2L.ai website [16] to download the code and then unzip it. Alternatively, you can fetch the notebooks from the command line as follows:

```
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en-1.0.3.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
cd pytorch
```

If you do not already have `unzip` installed, first run `sudo apt-get install unzip`. Now we can start the Jupyter Notebook server by running:

```
jupyter notebook
```

At this point, you can open http://localhost:8888 (it may have already opened automatically) in your web browser. Then we can run the code for each section of the book. Whenever you open a new command line window, you will need to execute `conda activate d2l` to activate the runtime environment before running the D2L notebooks, or updating your packages (either the deep learning framework or the `d2l` package). To exit the environment, run `conda deactivate`.

Discussions[17].

17 

Notation

Throughout this book, we adhere to the following notational conventions. Note that some of these symbols are placeholders, while others refer to specific objects. As a general rule of thumb, the indefinite article "a" often indicates that the symbol is a placeholder and that similarly formatted symbols can denote other objects of the same type. For example, "x: a scalar" means that lowercased letters generally represent scalar values, but "$\mathbb{Z}$: the set of integers" refers specifically to the symbol $\mathbb{Z}$.

Numerical Objects

- x: a scalar
- $\mathbf{x}$: a vector
- $\mathbf{X}$: a matrix
- X: a general tensor
- $\mathbf{I}$: the identity matrix (of some given dimension), i.e., a square matrix with 1 on all diagonal entries and 0 on all off-diagonals
- x_i, $[\mathbf{x}]_i$: the i^{th} element of vector $\mathbf{x}$
- x_{ij}, $x_{i,j}$,$[\mathbf{X}]_{ij}$, $[\mathbf{X}]_{i,j}$: the element of matrix $\mathbf{X}$ at row i and column j.

Set Theory

- $\mathcal{X}$: a set
- $\mathbb{Z}$: the set of integers
- $\mathbb{Z}^+$: the set of positive integers
- $\mathbb{R}$: the set of real numbers
- $\mathbb{R}^n$: the set of n-dimensional vectors of real numbers

- $\mathbb{R}^{a \times b}$: The set of matrices of real numbers with a rows and b columns
- $|\mathcal{X}|$: cardinality (number of elements) of set $\mathcal{X}$
- $\mathcal{A} \cup \mathcal{B}$: union of sets $\mathcal{A}$ and $\mathcal{B}$
- $\mathcal{A} \cap \mathcal{B}$: intersection of sets $\mathcal{A}$ and $\mathcal{B}$
- $\mathcal{A} \setminus \mathcal{B}$: set subtraction of $\mathcal{B}$ from $\mathcal{A}$ (contains only those elements of $\mathcal{A}$ that do not belong to $\mathcal{B}$)

Functions and Operators

- $f(\cdot)$: a function
- $\log(\cdot)$: the natural logarithm (base e)
- $\log_2(\cdot)$: logarithm to base 2
- $\exp(\cdot)$: the exponential function
- $\mathbf{1}(\cdot)$: the indicator function; evaluates to 1 if the boolean argument is true, and 0 otherwise
- $\mathbf{1}_{\mathcal{X}}(z)$: the set-membership indicator function; evaluates to 1 if the element z belongs to the set $\mathcal{X}$ and 0 otherwise
- $(\cdot)^\top$: transpose of a vector or a matrix
- $\mathbf{X}^{-1}$: inverse of matrix $\mathbf{X}$
- $\odot$: Hadamard (elementwise) product
- $[\cdot, \cdot]$: concatenation
- $\|\cdot\|_p$: ℓ_p norm
- $\|\cdot\|$: ℓ_2 norm
- $\langle \mathbf{x}, \mathbf{y} \rangle$: inner (dot) product of vectors $\mathbf{x}$ and $\mathbf{y}$
- $\sum$: summation over a collection of elements
- $\prod$: product over a collection of elements
- $\stackrel{\text{def}}{=}$: an equality asserted as a definition of the symbol on the left-hand side

Calculus

- $\frac{dy}{dx}$: derivative of y with respect to x
- $\frac{\partial y}{\partial x}$: partial derivative of y with respect to x
- $\nabla_{\mathbf{x}} y$: gradient of y with respect to $\mathbf{x}$
- $\int_a^b f(x)\, dx$: definite integral of f from a to b with respect to x
- $\int f(x)\, dx$: indefinite integral of f with respect to x

Probability and Information Theory

- X: a random variable
- P: a probability distribution
- $X \sim P$: the random variable X follows distribution P
- $P(X = x)$: the probability assigned to the event where random variable X takes value x
- $P(X \mid Y)$: the conditional probability distribution of X given Y
- $p(\cdot)$: a probability density function (PDF) associated with distribution P
- $E[X]$: expectation of a random variable X
- $X \perp Y$: random variables X and Y are independent
- $X \perp Y \mid Z$: random variables X and Y are conditionally independent given Z
- σ_X: standard deviation of random variable X
- $\text{Var}(X)$: variance of random variable X, equal to σ_X^2
- $\text{Cov}(X, Y)$: covariance of random variables X and Y
- $\rho(X, Y)$: the Pearson correlation coefficient between X and Y, equals $\frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y}$
- $H(X)$: entropy of random variable X
- $D_{\text{KL}}(P\|Q)$: the KL-divergence (or relative entropy) from distribution Q to distribution P

Discussions[18].

18 

1 Introduction

Until recently, nearly every computer program that you might have interacted with during an ordinary day was coded up as a rigid set of rules specifying precisely how it should behave. Say that we wanted to write an application to manage an e-commerce platform. After huddling around a whiteboard for a few hours to ponder the problem, we might settle on the broad strokes of a working solution, for example: (i) users interact with the application through an interface running in a web browser or mobile application; (ii) our application interacts with a commercial-grade database engine to keep track of each user's state and maintain records of historical transactions; and (iii) at the heart of our application, the *business logic* (you might say, the *brains*) of our application spells out a set of rules that map every conceivable circumstance to the corresponding action that our program should take.

To build the brains of our application, we might enumerate all the common events that our program should handle. For example, whenever a customer clicks to add an item to their shopping cart, our program should add an entry to the shopping cart database table, associating that user's ID with the requested product's ID. We might then attempt to step through every possible corner case, testing the appropriateness of our rules and making any necessary modifications. What happens if a user initiates a purchase with an empty cart? While few developers ever get it completely right the first time (it might take some test runs to work out the kinks), for the most part we can write such programs and confidently launch them *before* ever seeing a real customer. Our ability to manually design automated systems that drive functioning products and systems, often in novel situations, is a remarkable cognitive feat. And when you are able to devise solutions that work 100% of the time, you typically should not be worrying about machine learning.

Fortunately for the growing community of machine learning scientists, many tasks that we would like to automate do not bend so easily to human ingenuity. Imagine huddling around the whiteboard with the smartest minds you know, but this time you are tackling one of the following problems:

- Write a program that predicts tomorrow's weather given geographic information, satellite images, and a trailing window of past weather.
- Write a program that takes in a factoid question, expressed in free-form text, and answers it correctly.
- Write a program that, given an image, identifies every person depicted in it and draws outlines around each.

- Write a program that presents users with products that they are likely to enjoy but unlikely, in the natural course of browsing, to encounter.

For these problems, even elite programmers would struggle to code up solutions from scratch. The reasons can vary. Sometimes the program that we are looking for follows a pattern that changes over time, so there is no fixed right answer! In such cases, any successful solution must adapt gracefully to a changing world. At other times, the relationship (say between pixels, and abstract categories) may be too complicated, requiring thousands or millions of computations and following unknown principles. In the case of image recognition, the precise steps required to perform the task lie beyond our conscious understanding, even though our subconscious cognitive processes execute the task effortlessly.

Machine learning is the study of algorithms that can learn from experience. As a machine learning algorithm accumulates more experience, typically in the form of observational data or interactions with an environment, its performance improves. Contrast this with our deterministic e-commerce platform, which follows the same business logic, no matter how much experience accrues, until the developers themselves learn and decide that it is time to update the software. In this book, we will teach you the fundamentals of machine learning, focusing in particular on *deep learning*, a powerful set of techniques driving innovations in areas as diverse as computer vision, natural language processing, healthcare, and genomics.

1.1 A Motivating Example

Before beginning writing, the authors of this book, like much of the work force, had to become caffeinated. We hopped in the car and started driving. Using an iPhone, Alex called out "Hey Siri", awakening the phone's voice recognition system. Then Mu commanded "directions to Blue Bottle coffee shop". The phone quickly displayed the transcription of his command. It also recognized that we were asking for directions and launched the Maps application (app) to fulfill our request. Once launched, the Maps app identified a number of routes. Next to each route, the phone displayed a predicted transit time. While this story was fabricated for pedagogical convenience, it demonstrates that in the span of just a few seconds, our everyday interactions with a smart phone can engage several machine learning models.

Imagine just writing a program to respond to a *wake word* such as "Alexa", "OK Google", and "Hey Siri". Try coding it up in a room by yourself with nothing but a computer and a code editor, as illustrated in Fig. 1.1.1. How would you write such a program from first principles? Think about it… the problem is hard. Every second, the microphone will collect roughly 44,000 samples. Each sample is a measurement of the amplitude of the sound wave. What rule could map reliably from a snippet of raw audio to confident predictions $\{\text{yes}, \text{no}\}$ about whether the snippet contains the wake word? If you are stuck, do not worry.

We do not know how to write such a program from scratch either. That is why we use machine learning.

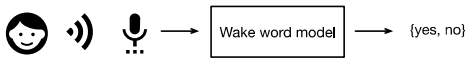


Fig. 1.1.1 Identify a wake word.

Here is the trick. Often, even when we do not know how to tell a computer explicitly how to map from inputs to outputs, we are nonetheless capable of performing the cognitive feat ourselves. In other words, even if you do not know how to program a computer to recognize the word "Alexa", you yourself are able to recognize it. Armed with this ability, we can collect a huge *dataset* containing examples of audio snippets and associated labels, indicating which snippets contain the wake word. In the currently dominant approach to machine learning, we do not attempt to design a system *explicitly* to recognize wake words. Instead, we define a flexible program whose behavior is determined by a number of *parameters*. Then we use the dataset to determine the best possible parameter values, i.e., those that improve the performance of our program with respect to a chosen performance measure.

You can think of the parameters as knobs that we can turn, manipulating the behavior of the program. Once the parameters are fixed, we call the program a *model*. The set of all distinct programs (input–output mappings) that we can produce just by manipulating the parameters is called a *family* of models. And the "meta-program" that uses our dataset to choose the parameters is called a *learning algorithm*.

Before we can go ahead and engage the learning algorithm, we have to define the problem precisely, pinning down the exact nature of the inputs and outputs, and choosing an appropriate model family. In this case, our model receives a snippet of audio as *input*, and the model generates a selection among $\{\text{yes}, \text{no}\}$ as *output*. If all goes according to plan the model's guesses will typically be correct as to whether the snippet contains the wake word.

If we choose the right family of models, there should exist one setting of the knobs such that the model fires "yes" every time it hears the word "Alexa". Because the exact choice of the wake word is arbitrary, we will probably need a model family sufficiently rich that, via another setting of the knobs, it could fire "yes" only upon hearing the word "Apricot". We expect that the same model family should be suitable for "Alexa" recognition and "Apricot" recognition because they seem, intuitively, to be similar tasks. However, we might need a different family of models entirely if we want to deal with fundamentally different inputs or outputs, say if we wanted to map from images to captions, or from English sentences to Chinese sentences.

As you might guess, if we just set all of the knobs randomly, it is unlikely that our model will recognize "Alexa", "Apricot", or any other English word. In machine learning, the *learning* is the process by which we discover the right setting of the knobs for coercing the

desired behavior from our model. In other words, we *train* our model with data. As shown in Fig. 1.1.2, the training process usually looks like the following:

1. Start off with a randomly initialized model that cannot do anything useful.
2. Grab some of your data (e.g., audio snippets and corresponding {yes, no} labels).
3. Tweak the knobs to make the model perform better as assessed on those examples.
4. Repeat Steps 2 and 3 until the model is awesome.

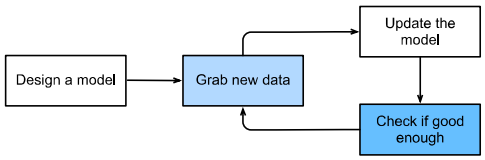


Fig. 1.1.2 A typical training process.

To summarize, rather than code up a wake word recognizer, we code up a program that can *learn* to recognize wake words, if presented with a large labeled dataset. You can think of this act of determining a program's behavior by presenting it with a dataset as *programming with data*. That is to say, we can "program" a cat detector by providing our machine learning system with many examples of cats and dogs. This way the detector will eventually learn to emit a very large positive number if it is a cat, a very large negative number if it is a dog, and something closer to zero if it is not sure. This barely scratches the surface of what machine learning can do. Deep learning, which we will explain in greater detail later, is just one among many popular methods for solving machine learning problems.

1.2 Key Components

In our wake word example, we described a dataset consisting of audio snippets and binary labels, and we gave a hand-wavy sense of how we might train a model to approximate a mapping from snippets to classifications. This sort of problem, where we try to predict a designated unknown label based on known inputs given a dataset consisting of examples for which the labels are known, is called *supervised learning*. This is just one among many kinds of machine learning problems. Before we explore other varieties, we would like to shed more light on some core components that will follow us around, no matter what kind of machine learning problem we tackle:

1. The *data* that we can learn from.
2. A *model* of how to transform the data.
3. An *objective function* that quantifies how well (or badly) the model is doing.
4. An *algorithm* to adjust the model's parameters to optimize the objective function.

1.2.1 Data

It might go without saying that you cannot do data science without data. We could lose hundreds of pages pondering what precisely data *is*, but for now, we will focus on the key properties of the datasets that we will be concerned with. Generally, we are concerned with a collection of examples. In order to work with data usefully, we typically need to come up with a suitable numerical representation. Each *example* (or *data point*, *data instance*, *sample*) typically consists of a set of attributes called *features* (sometimes called *covariates* or *inputs*), based on which the model must make its predictions. In supervised learning problems, our goal is to predict the value of a special attribute, called the *label* (or *target*), that is not part of the model's input.

If we were working with image data, each example might consist of an individual photograph (the features) and a number indicating the category to which the photograph belongs (the label). The photograph would be represented numerically as three grids of numerical values representing the brightness of red, green, and blue light at each pixel location. For example, a 200×200 pixel color photograph would consist of $200 \times 200 \times 3 = 120000$ numerical values.

Alternatively, we might work with electronic health record data and tackle the task of predicting the likelihood that a given patient will survive the next 30 days. Here, our features might consist of a collection of readily available attributes and frequently recorded measurements, including age, vital signs, comorbidities, current medications, and recent procedures. The label available for training would be a binary value indicating whether each patient in the historical data survived within the 30-day window.

In such cases, when every example is characterized by the same number of numerical features, we say that the inputs are fixed-length vectors and we call the (constant) length of the vectors the *dimensionality* of the data. As you might imagine, fixed-length inputs can be convenient, giving us one less complication to worry about. However, not all data can easily be represented as *fixed-length* vectors. While we might expect microscope images to come from standard equipment, we cannot expect images mined from the Internet all to have the same resolution or shape. For images, we might consider cropping them to a standard size, but that strategy only gets us so far. We risk losing information in the cropped-out portions. Moreover, text data resists fixed-length representations even more stubbornly. Consider the customer reviews left on e-commerce sites such as Amazon, IMDb, and TripAdvisor. Some are short: "it stinks!". Others ramble for pages. One major advantage of deep learning over traditional methods is the comparative grace with which modern models can handle *varying-length* data.

Generally, the more data we have, the easier our job becomes. When we have more data, we can train more powerful models and rely less heavily on preconceived assumptions. The regime change from (comparatively) small to big data is a major contributor to the success of modern deep learning. To drive the point home, many of the most exciting models in deep learning do not work without large datasets. Some others might work in the small data regime, but are no better than traditional approaches.

Finally, it is not enough to have lots of data and to process it cleverly. We need the *right*

data. If the data is full of mistakes, or if the chosen features are not predictive of the target quantity of interest, learning is going to fail. The situation is captured well by the cliché: *garbage in, garbage out*. Moreover, poor predictive performance is not the only potential consequence. In sensitive applications of machine learning, like predictive policing, resume screening, and risk models used for lending, we must be especially alert to the consequences of garbage data. One commonly occurring failure mode concerns datasets where some groups of people are unrepresented in the training data. Imagine applying a skin cancer recognition system that had never seen black skin before. Failure can also occur when the data does not only under-represent some groups but reflects societal prejudices. For example, if past hiring decisions are used to train a predictive model that will be used to screen resumes then machine learning models could inadvertently capture and automate historical injustices. Note that this can all happen without the data scientist actively conspiring, or even being aware.

1.2.2 Models

Most machine learning involves transforming the data in some sense. We might want to build a system that ingests photos and predicts smiley-ness. Alternatively, we might want to ingest a set of sensor readings and predict how normal vs. anomalous the readings are. By *model*, we denote the computational machinery for ingesting data of one type, and spitting out predictions of a possibly different type. In particular, we are interested in *statistical models* that can be estimated from data. While simple models are perfectly capable of addressing appropriately simple problems, the problems that we focus on in this book stretch the limits of classical methods. Deep learning is differentiated from classical approaches principally by the set of powerful models that it focuses on. These models consist of many successive transformations of the data that are chained together top to bottom, thus the name *deep learning*. On our way to discussing deep models, we will also discuss some more traditional methods.

1.2.3 Objective Functions

Earlier, we introduced machine learning as learning from experience. By *learning* here, we mean improving at some task over time. But who is to say what constitutes an improvement? You might imagine that we could propose updating our model, and some people might disagree on whether our proposal constituted an improvement or not.

In order to develop a formal mathematical system of learning machines, we need to have formal measures of how good (or bad) our models are. In machine learning, and optimization more generally, we call these *objective functions*. By convention, we usually define objective functions so that lower is better. This is merely a convention. You can take any function for which higher is better, and turn it into a new function that is qualitatively identical but for which lower is better by flipping the sign. Because we choose lower to be better, these functions are sometimes called *loss functions*.

When trying to predict numerical values, the most common loss function is *squared error*, i.e., the square of the difference between the prediction and the ground truth target. For classification, the most common objective is to minimize error rate, i.e., the fraction of

examples on which our predictions disagree with the ground truth. Some objectives (e.g., squared error) are easy to optimize, while others (e.g., error rate) are difficult to optimize directly, owing to non-differentiability or other complications. In these cases, it is common instead to optimize a *surrogate objective*.

During optimization, we think of the loss as a function of the model's parameters, and treat the training dataset as a constant. We learn the best values of our model's parameters by minimizing the loss incurred on a set consisting of some number of examples collected for training. However, doing well on the training data does not guarantee that we will do well on unseen data. So we will typically want to split the available data into two partitions: the *training dataset* (or *training set*), for learning model parameters; and the *test dataset* (or *test set*), which is held out for evaluation. At the end of the day, we typically report how our models perform on both partitions. You could think of training performance as analogous to the scores that a student achieves on the practice exams used to prepare for some real final exam. Even if the results are encouraging, that does not guarantee success on the final exam. Over the course of studying, the student might begin to memorize the practice questions, appearing to master the topic but faltering when faced with previously unseen questions on the actual final exam. When a model performs well on the training set but fails to generalize to unseen data, we say that it is *overfitting* to the training data.

1.2.4 Optimization Algorithms

Once we have got some data source and representation, a model, and a well-defined objective function, we need an algorithm capable of searching for the best possible parameters for minimizing the loss function. Popular optimization algorithms for deep learning are based on an approach called *gradient descent*. In brief, at each step, this method checks to see, for each parameter, how that training set loss would change if you perturbed that parameter by just a small amount. It would then update the parameter in the direction that lowers the loss.

1.3 Kinds of Machine Learning Problems

The wake word problem in our motivating example is just one among many that machine learning can tackle. To motivate the reader further and provide us with some common language that will follow us throughout the book, we now provide a broad overview of the landscape of machine learning problems.

1.3.1 Supervised Learning

Supervised learning describes tasks where we are given a dataset containing both features and labels and asked to produce a model that predicts the labels when given input features. Each feature–label pair is called an example. Sometimes, when the context is clear, we may use the term *examples* to refer to a collection of inputs, even when the corresponding

labels are unknown. The supervision comes into play because, for choosing the parameters, we (the supervisors) provide the model with a dataset consisting of labeled examples. In probabilistic terms, we typically are interested in estimating the conditional probability of a label given input features. While it is just one among several paradigms, supervised learning accounts for the majority of successful applications of machine learning in industry. Partly that is because many important tasks can be described crisply as estimating the probability of something unknown given a particular set of available data:

- Predict cancer vs. not cancer, given a computer tomography image.
- Predict the correct translation in French, given a sentence in English.
- Predict the price of a stock next month based on this month's financial reporting data.

While all supervised learning problems are captured by the simple description "predicting the labels given input features", supervised learning itself can take diverse forms and require tons of modeling decisions, depending on (among other considerations) the type, size, and quantity of the inputs and outputs. For example, we use different models for processing sequences of arbitrary lengths and fixed-length vector representations. We will visit many of these problems in depth throughout this book.

Informally, the learning process looks something like the following. First, grab a big collection of examples for which the features are known and select from them a random subset, acquiring the ground truth labels for each. Sometimes these labels might be available data that have already been collected (e.g., did a patient die within the following year?) and other times we might need to employ human annotators to label the data, (e.g., assigning images to categories). Together, these inputs and corresponding labels comprise the training set. We feed the training dataset into a supervised learning algorithm, a function that takes as input a dataset and outputs another function: the learned model. Finally, we can feed previously unseen inputs to the learned model, using its outputs as predictions of the corresponding label. The full process is drawn in Fig. 1.3.1.

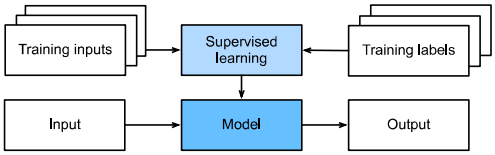


Fig. 1.3.1 Supervised learning.

Regression

Perhaps the simplest supervised learning task to wrap your head around is *regression*. Consider, for example, a set of data harvested from a database of home sales. We might construct a table, in which each row corresponds to a different house, and each column corresponds to some relevant attribute, such as the square footage of a house, the number of bedrooms, the number of bathrooms, and the number of minutes (walking) to the center of town. In this dataset, each example would be a specific house, and the corresponding

feature vector would be one row in the table. If you live in New York or San Francisco, and you are not the CEO of Amazon, Google, Microsoft, or Facebook, the (sq. footage, no. of bedrooms, no. of bathrooms, walking distance) feature vector for your home might look something like: $[600, 1, 1, 60]$. However, if you live in Pittsburgh, it might look more like $[3000, 4, 3, 10]$. Fixed-length feature vectors like this are essential for most classic machine learning algorithms.

What makes a problem a regression is actually the form of the target. Say that you are in the market for a new home. You might want to estimate the fair market value of a house, given some features such as above. The data here might consist of historical home listings and the labels might be the observed sales prices. When labels take on arbitrary numerical values (even within some interval), we call this a *regression* problem. The goal is to produce a model whose predictions closely approximate the actual label values.

Lots of practical problems are easily described as regression problems. Predicting the rating that a user will assign to a movie can be thought of as a regression problem and if you designed a great algorithm to accomplish this feat in 2009, you might have won the 1-million-dollar Netflix prize[19]. Predicting the length of stay for patients in the hospital is
19  also a regression problem. A good rule of thumb is that any *how much?* or *how many?* problem is likely to be regression. For example:

- How many hours will this surgery take?
- How much rainfall will this town have in the next six hours?

Even if you have never worked with machine learning before, you have probably worked through a regression problem informally. Imagine, for example, that you had your drains repaired and that your contractor spent 3 hours removing gunk from your sewage pipes. Then they sent you a bill of 350 dollars. Now imagine that your friend hired the same contractor for 2 hours and received a bill of 250 dollars. If someone then asked you how much to expect on their upcoming gunk-removal invoice you might make some reasonable assumptions, such as more hours worked costs more dollars. You might also assume that there is some base charge and that the contractor then charges per hour. If these assumptions held true, then given these two data examples, you could already identify the contractor's pricing structure: 100 dollars per hour plus 50 dollars to show up at your house. If you followed that much, then you already understand the high-level idea behind *linear* regression.

In this case, we could produce the parameters that exactly matched the contractor's prices. Sometimes this is not possible, e.g., if some of the variation arises from factors beyond your two features. In these cases, we will try to learn models that minimize the distance between our predictions and the observed values. In most of our chapters, we will focus on minimizing the squared error loss function. As we will see later, this loss corresponds to the assumption that our data were corrupted by Gaussian noise.

Classification

While regression models are great for addressing *how many?* questions, lots of problems do not fit comfortably in this template. Consider, for example, a bank that wants to develop a

check scanning feature for its mobile app. Ideally, the customer would simply snap a photo of a check and the app would automatically recognize the text from the image. Assuming that we had some ability to segment out image patches corresponding to each handwritten character, then the primary remaining task would be to determine which character among some known set is depicted in each image patch. These kinds of *which one?* problems are called *classification* and require a different set of tools from those used for regression, although many techniques will carry over.

In *classification*, we want our model to look at features, e.g., the pixel values in an image, and then predict to which *category* (sometimes called a *class*) among some discrete set of options, an example belongs. For handwritten digits, we might have ten classes, corresponding to the digits 0 through 9. The simplest form of classification is when there are only two classes, a problem which we call *binary classification*. For example, our dataset could consist of images of animals and our labels might be the classes {cat, dog}. Whereas in regression we sought a regressor to output a numerical value, in classification we seek a classifier, whose output is the predicted class assignment.

For reasons that we will get into as the book gets more technical, it can be difficult to optimize a model that can only output a *firm* categorical assignment, e.g., either "cat" or "dog". In these cases, it is usually much easier to express our model in the language of probabilities. Given features of an example, our model assigns a probability to each possible class. Returning to our animal classification example where the classes are {cat, dog}, a classifier might see an image and output the probability that the image is a cat as 0.9. We can interpret this number by saying that the classifier is 90% sure that the image depicts a cat. The magnitude of the probability for the predicted class conveys a notion of uncertainty. It is not the only one available and we will discuss others in chapters dealing with more advanced topics.

When we have more than two possible classes, we call the problem *multiclass classification*. Common examples include handwritten character recognition {0, 1, 2, ... 9, a, b, c, ...}. While we attacked regression problems by trying to minimize the squared error loss function, the common loss function for classification problems is called *cross-entropy*, whose name will be demystified when we introduce information theory in later chapters.

Note that the most likely class is not necessarily the one that you are going to use for your decision. Assume that you find a beautiful mushroom in your backyard as shown in Fig. 1.3.2.

Now, assume that you built a classifier and trained it to predict whether a mushroom is poisonous based on a photograph. Say our poison-detection classifier outputs that the probability that Fig. 1.3.2 shows a death cap is 0.2. In other words, the classifier is 80% sure that our mushroom is not a death cap. Still, you would have to be a fool to eat it. That is because the certain benefit of a delicious dinner is not worth a 20% risk of dying from it. In other words, the effect of the uncertain risk outweighs the benefit by far. Thus, in order to make a decision about whether to eat the mushroom, we need to compute the expected detriment associated with each action which depends both on the likely outcomes and the benefits or harms associated with each. In this case, the detriment incurred by eating the mushroom

Fig. 1.3.2 Death cap - do not eat!

might be $0.2 \times \infty + 0.8 \times 0 = \infty$, whereas the loss of discarding it is $0.2 \times 0 + 0.8 \times 1 = 0.8$. Our caution was justified: as any mycologist would tell us, the mushroom in Fig. 1.3.2 is actually a death cap.

Classification can get much more complicated than just binary or multiclass classification. For instance, there are some variants of classification addressing hierarchically structured classes. In such cases not all errors are equal—if we must err, we might prefer to misclassify to a related class rather than a distant class. Usually, this is referred to as *hierarchical classification*. For inspiration, you might think of Linnaeus[20], who organized fauna in a hierarchy.

20

In the case of animal classification, it might not be so bad to mistake a poodle for a schnauzer, but our model would pay a huge penalty if it confused a poodle with a dinosaur. Which hierarchy is relevant might depend on how you plan to use the model. For example, rattlesnakes and garter snakes might be close on the phylogenetic tree, but mistaking a rattler for a garter could have fatal consequences.

Tagging

Some classification problems fit neatly into the binary or multiclass classification setups. For example, we could train a normal binary classifier to distinguish cats from dogs. Given the current state of computer vision, we can do this easily, with off-the-shelf tools. Nonetheless, no matter how accurate our model gets, we might find ourselves in trouble when the classifier encounters an image of the *Town Musicians of Bremen*, a popular German fairy tale featuring four animals (Fig. 1.3.3).

As you can see, the photo features a cat, a rooster, a dog, and a donkey, with some trees in the background. If we anticipate encountering such images, multiclass classification might not be the right problem formulation. Instead, we might want to give the model the option of saying the image depicts a cat, a dog, a donkey, *and* a rooster.

Fig. 1.3.3 A donkey, a dog, a cat, and a rooster.

The problem of learning to predict classes that are not mutually exclusive is called *multi-label classification*. Auto-tagging problems are typically best described in terms of multi-label classification. Think of the tags people might apply to posts on a technical blog, e.g., "machine learning", "technology", "gadgets", "programming languages", "Linux", "cloud computing", "AWS". A typical article might have 5–10 tags applied. Typically, tags will exhibit some correlation structure. Posts about "cloud computing" are likely to mention "AWS" and posts about "machine learning" are likely to mention "GPUs".

Sometimes such tagging problems draw on enormous label sets. The National Library of Medicine employs many professional annotators who associate each article to be indexed in PubMed with a set of tags drawn from the Medical Subject Headings (MeSH) ontology, a collection of roughly 28,000 tags. Correctly tagging articles is important because it allows researchers to conduct exhaustive reviews of the literature. This is a time-consuming process and typically there is a one-year lag between archiving and tagging. Machine learning can provide provisional tags until each article has a proper manual review. Indeed, for several years, the BioASQ organization has hosted competitions[21] for this task.

21

Search

In the field of information retrieval, we often impose ranks on sets of items. Take web search for example. The goal is less to determine *whether* a particular page is relevant for a query, but rather which, among a set of relevant results, should be shown most prominently to a particular user. One way of doing this might be to first assign a score to every element in the set and then to retrieve the top-rated elements. PageRank [22], the original secret sauce behind the Google search engine, was an early example of such a scoring system. Weirdly, the scoring provided by PageRank did not depend on the actual query. Instead, they relied on a simple relevance filter to identify the set of relevant candidates and then used PageRank to prioritize the more authoritative pages. Nowadays, search engines use machine learning and behavioral models to obtain query-dependent relevance scores. There are entire academic conferences devoted to this subject.

22 

Recommender Systems

Recommender systems are another problem setting that is related to search and ranking. The problems are similar insofar as the goal is to display a set of items relevant to the user. The main difference is the emphasis on *personalization* to specific users in the context of recommender systems. For instance, for movie recommendations, the results page for a science fiction fan and the results page for a connoisseur of Peter Sellers comedies might differ significantly. Similar problems pop up in other recommendation settings, e.g., for retail products, music, and news recommendation.

In some cases, customers provide explicit feedback, communicating how much they liked a particular product (e.g., the product ratings and reviews on Amazon, IMDb, or Goodreads). In other cases, they provide implicit feedback, e.g., by skipping titles on a playlist, which might indicate dissatisfaction or maybe just indicate that the song was inappropriate in context. In the simplest formulations, these systems are trained to estimate some score, such as an expected star rating or the probability that a given user will purchase a particular item.

Given such a model, for any given user, we could retrieve the set of objects with the largest scores, which could then be recommended to the user. Production systems are considerably more advanced and take detailed user activity and item characteristics into account when computing such scores. Fig. 1.3.4 displays the deep learning books recommended by Amazon based on personalization algorithms tuned to capture Aston's preferences.

Despite their tremendous economic value, recommender systems naively built on top of predictive models suffer some serious conceptual flaws. To start, we only observe *censored feedback*: users preferentially rate movies that they feel strongly about. For example, on a five-point scale, you might notice that items receive many one- and five-star ratings but that there are conspicuously few three-star ratings. Moreover, current purchase habits are often a result of the recommendation algorithm currently in place, but learning algorithms do not always take this detail into account. Thus it is possible for feedback loops to form where a recommender system preferentially pushes an item that is then taken to be better (due to greater purchases) and in turn is recommended even more frequently. Many of

Fig. 1.3.4 Deep learning books recommended by Amazon.

these problems—about how to deal with censoring, incentives, and feedback loops—are important open research questions.

Sequence Learning

So far, we have looked at problems where we have some fixed number of inputs and produce a fixed number of outputs. For example, we considered predicting house prices given a fixed set of features: square footage, number of bedrooms, number of bathrooms, and the transit time to downtown. We also discussed mapping from an image (of fixed dimension) to the predicted probabilities that it belongs to each among a fixed number of classes and predicting star ratings associated with purchases based on the user ID and product ID alone. In these cases, once our model is trained, after each test example is fed into our model, it is immediately forgotten. We assumed that successive observations were independent and thus there was no need to hold on to this context.

But how should we deal with video snippets? In this case, each snippet might consist of a different number of frames. And our guess of what is going on in each frame might be much stronger if we take into account the previous or succeeding frames. The same goes for language. For example, one popular deep learning problem is machine translation: the task of ingesting sentences in some source language and predicting their translations in another language.

Such problems also occur in medicine. We might want a model to monitor patients in the intensive care unit and to fire off alerts whenever their risk of dying in the next 24 hours exceeds some threshold. Here, we would not throw away everything that we know about

the patient history every hour, because we might not want to make predictions based only on the most recent measurements.

Questions like these are among the most exciting applications of machine learning and they are instances of *sequence learning*. They require a model either to ingest sequences of inputs or to emit sequences of outputs (or both). Specifically, *sequence-to-sequence learning* considers problems where both inputs and outputs consist of variable-length sequences. Examples include machine translation and speech-to-text transcription. While it is impossible to consider all types of sequence transformations, the following special cases are worth mentioning.

Tagging and Parsing. This involves annotating a text sequence with attributes. Here, the inputs and outputs are *aligned*, i.e., they are of the same number and occur in a corresponding order. For instance, in *part-of-speech (PoS) tagging*, we annotate every word in a sentence with the corresponding part of speech, i.e., "noun" or "direct object". Alternatively, we might want to know which groups of contiguous words refer to named entities, like *people*, *places*, or *organizations*. In the cartoonishly simple example below, we might just want to indicate whether or not any word in the sentence is part of a named entity (tagged as "Ent").

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

Automatic Speech Recognition. With speech recognition, the input sequence is an audio recording of a speaker (Fig. 1.3.5), and the output is a transcript of what the speaker said. The challenge is that there are many more audio frames (sound is typically sampled at 8kHz or 16kHz) than text, i.e., there is no 1:1 correspondence between audio and text, since thousands of samples may correspond to a single spoken word. These are sequence-to-sequence learning problems, where the output is much shorter than the input. While humans are remarkably good at recognizing speech, even from low-quality audio, getting computers to perform the same feat is a formidable challenge.

Fig. 1.3.5 `-D-e-e-p- L-ea-r-ni-ng-` in an audio recording.

Text to Speech. This is the inverse of automatic speech recognition. Here, the input is text and the output is an audio file. In this case, the output is much longer than the input.

Machine Translation. Unlike the case of speech recognition, where corresponding inputs and outputs occur in the same order, in machine translation, unaligned data poses a new challenge. Here the input and output sequences can have different lengths, and the corre-

sponding regions of the respective sequences may appear in a different order. Consider the following illustrative example of the peculiar tendency of Germans to place the verbs at the end of sentences:

```
German:           Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
English:          Have you already looked at this excellent textbook?
Wrong alignment:  Have you yourself already this excellent textbook looked at?
```

Many related problems pop up in other learning tasks. For instance, determining the order in which a user reads a webpage is a two-dimensional layout analysis problem. Dialogue problems exhibit all kinds of additional complications, where determining what to say next requires taking into account real-world knowledge and the prior state of the conversation across long temporal distances. Such topics are active areas of research.

1.3.2 Unsupervised and Self-Supervised Learning

The previous examples focused on supervised learning, where we feed the model a giant dataset containing both the features and corresponding label values. You could think of the supervised learner as having an extremely specialized job and an extremely dictatorial boss. The boss stands over the learner's shoulder and tells them exactly what to do in every situation until they learn to map from situations to actions. Working for such a boss sounds pretty lame. On the other hand, pleasing such a boss is pretty easy. You just recognize the pattern as quickly as possible and imitate the boss's actions.

Considering the opposite situation, it could be frustrating to work for a boss who has no idea what they want you to do. However, if you plan to be a data scientist, you had better get used to it. The boss might just hand you a giant dump of data and tell you to *do some data science with it!* This sounds vague because it is vague. We call this class of problems *unsupervised learning*, and the type and number of questions we can ask is limited only by our creativity. We will address unsupervised learning techniques in later chapters. To whet your appetite for now, we describe a few of the following questions you might ask.

- Can we find a small number of prototypes that accurately summarize the data? Given a set of photos, can we group them into landscape photos, pictures of dogs, babies, cats, and mountain peaks? Likewise, given a collection of users' browsing activities, can we group them into users with similar behavior? This problem is typically known as *clustering*.

- Can we find a small number of parameters that accurately capture the relevant properties of the data? The trajectories of a ball are well described by velocity, diameter, and mass of the ball. Tailors have developed a small number of parameters that describe human body shape fairly accurately for the purpose of fitting clothes. These problems are referred to as *subspace estimation*. If the dependence is linear, it is called *principal component analysis*.

- Is there a representation of (arbitrarily structured) objects in Euclidean space such that symbolic properties can be well matched? This can be used to describe entities and their relations, such as "Rome" − "Italy" + "France" = "Paris".

- Is there a description of the root causes of much of the data that we observe? For instance, if we have demographic data about house prices, pollution, crime, location, education, and salaries, can we discover how they are related simply based on empirical data? The fields concerned with *causality* and *probabilistic graphical models* tackle such questions.
- Another important and exciting recent development in unsupervised learning is the advent of *deep generative models*. These models estimate the density of the data, either explicitly or *implicitly*. Once trained, we can use a generative model either to score examples according to how likely they are, or to sample synthetic examples from the learned distribution. Early deep learning breakthroughs in generative modeling came with the invention of *variational autoencoders* (Kingma and Welling, 2014, Rezende *et al.*, 2014) and continued with the development of *generative adversarial networks* (Goodfellow *et al.*, 2014). More recent advances include normalizing flows (Dinh *et al.*, 2014, Dinh *et al.*, 2017) and diffusion models (Ho *et al.*, 2020, Sohl-Dickstein *et al.*, 2015, Song and Ermon, 2019, Song *et al.*, 2021).

A further development in unsupervised learning has been the rise of *self-supervised learning*, techniques that leverage some aspect of the unlabeled data to provide supervision. For text, we can train models to "fill in the blanks" by predicting randomly masked words using their surrounding words (contexts) in big corpora without any labeling effort (Devlin *et al.*, 2018)! For images, we may train models to tell the relative position between two cropped regions of the same image (Doersch *et al.*, 2015), to predict an occluded part of an image based on the remaining portions of the image, or to predict whether two examples are perturbed versions of the same underlying image. Self-supervised models often learn representations that are subsequently leveraged by fine-tuning the resulting models on some downstream task of interest.

1.3.3 Interacting with an Environment

So far, we have not discussed where data actually comes from, or what actually happens when a machine learning model generates an output. That is because supervised learning and unsupervised learning do not address these issues in a very sophisticated way. In each case, we grab a big pile of data upfront, then set our pattern recognition machines in motion without ever interacting with the environment again. Because all the learning takes place after the algorithm is disconnected from the environment, this is sometimes called *offline learning*. For example, supervised learning assumes the simple interaction pattern depicted in Fig. 1.3.6.

This simplicity of offline learning has its charms. The upside is that we can worry about pattern recognition in isolation, with no concern about complications arising from interactions with a dynamic environment. But this problem formulation is limiting. If you grew up reading Asimov's Robot novels, then you probably picture artificially intelligent agents capable not only of making predictions, but also of taking actions in the world. We want to think about intelligent *agents*, not just predictive models. This means that we need to think about choosing *actions*, not just making predictions. In contrast to mere predictions, actions actually impact the environment. If we want to train an intelligent agent, we must

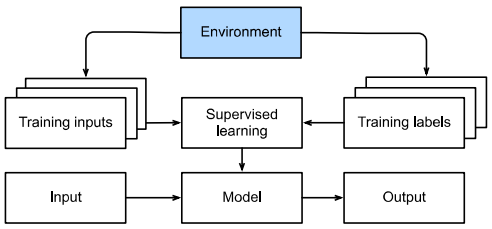


Fig. 1.3.6 Collecting data for supervised learning from an environment.

account for the way its actions might impact the future observations of the agent, and so offline learning is inappropriate.

Considering the interaction with an environment opens a whole set of new modeling questions. The following are just a few examples.

- Does the environment remember what we did previously?
- Does the environment want to help us, e.g., a user reading text into a speech recognizer?
- Does the environment want to beat us, e.g., spammers adapting their emails to evade spam filters?
- Does the environment have shifting dynamics? For example, would future data always resemble the past or would the patterns change over time, either naturally or in response to our automated tools?

These questions raise the problem of *distribution shift*, where training and test data are different. An example of this, that many of us may have met, is when taking exams written by a lecturer, while the homework was composed by their teaching assistants. Next, we briefly describe reinforcement learning, a rich framework for posing learning problems in which an agent interacts with an environment.

1.3.4 Reinforcement Learning

If you are interested in using machine learning to develop an agent that interacts with an environment and takes actions, then you are probably going to wind up focusing on *reinforcement learning*. This might include applications to robotics, to dialogue systems, and even to developing artificial intelligence (AI) for video games. *Deep reinforcement learning*, which applies deep learning to reinforcement learning problems, has surged in popularity. The breakthrough deep Q-network, that beat humans at Atari games using only the visual input (Mnih *et al.*, 2015), and the AlphaGo program, which dethroned the world champion at the board game Go (Silver *et al.*, 2016), are two prominent examples.

Reinforcement learning gives a very general statement of a problem in which an agent interacts with an environment over a series of time steps. At each time step, the agent receives some *observation* from the environment and must choose an *action* that is subsequently transmitted back to the environment via some mechanism (sometimes called an *actuator*), when, after each loop, the agent receives a reward from the environment. This process is

illustrated in Fig. 1.3.7. The agent then receives a subsequent observation, and chooses a subsequent action, and so on. The behavior of a reinforcement learning agent is governed by a *policy*. In brief, a *policy* is just a function that maps from observations of the environment to actions. The goal of reinforcement learning is to produce good policies.

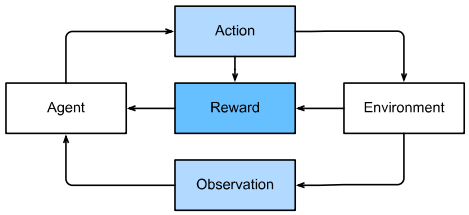


Fig. 1.3.7 The interaction between reinforcement learning and an environment.

It is hard to overstate the generality of the reinforcement learning framework. For example, supervised learning can be recast as reinforcement learning. Say we had a classification problem. We could create a reinforcement learning agent with one action corresponding to each class. We could then create an environment which gave a reward that was exactly equal to the loss function from the original supervised learning problem.

Further, reinforcement learning can also address many problems that supervised learning cannot. For example, in supervised learning, we always expect that the training input comes associated with the correct label. But in reinforcement learning, we do not assume that, for each observation the environment tells us the optimal action. In general, we just get some reward. Moreover, the environment may not even tell us which actions led to the reward.

Consider the game of chess. The only real reward signal comes at the end of the game when we either win, earning a reward of, say, 1, or when we lose, receiving a reward of, say, -1. So reinforcement learners must deal with the *credit assignment* problem: determining which actions to credit or blame for an outcome. The same goes for an employee who gets a promotion on October 11. That promotion likely reflects a number of well-chosen actions over the previous year. Getting promoted in the future requires figuring out which actions along the way led to the earlier promotions.

Reinforcement learners may also have to deal with the problem of partial observability. That is, the current observation might not tell you everything about your current state. Say your cleaning robot found itself trapped in one of many identical closets in your house. Rescuing the robot involves inferring its precise location which might require considering earlier observations prior to it entering the closet.

Finally, at any given point, reinforcement learners might know of one good policy, but there might be many other better policies that the agent has never tried. The reinforcement learner must constantly choose whether to *exploit* the best (currently) known strategy as a policy, or to *explore* the space of strategies, potentially giving up some short-term reward in exchange for knowledge.

The general reinforcement learning problem has a very general setting. Actions affect sub-

sequent observations. Rewards are only observed when they correspond to the chosen actions. The environment may be either fully or partially observed. Accounting for all this complexity at once may be asking too much. Moreover, not every practical problem exhibits all this complexity. As a result, researchers have studied a number of special cases of reinforcement learning problems.

When the environment is fully observed, we call the reinforcement learning problem a *Markov decision process*. When the state does not depend on the previous actions, we call it a *contextual bandit problem*. When there is no state, just a set of available actions with initially unknown rewards, we have the classic *multi-armed bandit problem*.

1.4 Roots

We have just reviewed a small subset of problems that machine learning can address. For a diverse set of machine learning problems, deep learning provides powerful tools for their solution. Although many deep learning methods are recent inventions, the core ideas behind learning from data have been studied for centuries. In fact, humans have held the desire to analyze data and to predict future outcomes for ages, and it is this desire that is at the root of much of natural science and mathematics. Two examples are the Bernoulli distribution, named after Jacob Bernoulli (1655–1705)[23], and the Gaussian distribution discovered by Carl Friedrich Gauss (1777–1855)[24]. Gauss invented, for instance, the least mean squares algorithm, which is still used today for a multitude of problems from insurance calculations to medical diagnostics. Such tools enhanced the experimental approach in the natural sciences—for instance, Ohm's law relating current and voltage in a resistor is perfectly described by a linear model.

Even in the middle ages, mathematicians had a keen intuition of estimates. For instance, the geometry book of Jacob Köbel (1460–1533)[25] illustrates averaging the length of 16 adult men's feet to estimate the typical foot length in the population (Fig. 1.4.1).

As a group of individuals exited a church, 16 adult men were asked to line up in a row and have their feet measured. The sum of these measurements was then divided by 16 to obtain an estimate for what now is called one foot. This "algorithm" was later improved to deal with misshapen feet; The two men with the shortest and longest feet were sent away, averaging only over the remainder. This is among the earliest examples of a trimmed mean estimate.

Statistics really took off with the availability and collection of data. One of its pioneers, Ronald Fisher (1890–1962)[26], contributed significantly to its theory and also its applications in genetics. Many of his algorithms (such as linear discriminant analysis) and concepts (such as the Fisher information matrix) still hold a prominent place in the foundations of modern statistics. Even his data resources had a lasting impact. The Iris dataset that Fisher released in 1936 is still sometimes used to demonstrate machine learning algorithms. Fisher was also a proponent of eugenics, which should remind us that the morally

Fig. 1.4.1 Estimating the length of a foot.

dubious use of data science has as long and enduring a history as its productive use in industry and the natural sciences.

Other influences for machine learning came from the information theory of Claude Shannon (1916–2001)[27] and the theory of computation proposed by Alan Turing (1912–1954)[28]. Turing posed the question "can machines think?" in his famous paper *Computing Machinery and Intelligence* (Turing, 1950). Describing what is now known as the Turing test, he proposed that a machine can be considered *intelligent* if it is difficult for a human evaluator to distinguish between the replies from a machine and those of a human, based purely on textual interactions.

27 

28 

Further influences came from neuroscience and psychology. After all, humans clearly exhibit intelligent behavior. Many scholars have asked whether one could explain and possibly reverse engineer this capacity. One of the first biologically inspired algorithms was formulated by Donald Hebb (1904–1985)[29]. In his groundbreaking book *The Organization of Behavior* (Hebb, 1949), he posited that neurons learn by positive reinforcement. This became known as the Hebbian learning rule. These ideas inspired later work, such as Rosenblatt's perceptron learning algorithm, and laid the foundations of many stochastic gradient descent algorithms that underpin deep learning today: reinforce desirable behavior and diminish undesirable behavior to obtain good settings of the parameters in a neural network.

29 

Biological inspiration is what gave *neural networks* their name. For over a century (dating back to the models of Alexander Bain, 1873, and James Sherrington, 1890), researchers have tried to assemble computational circuits that resemble networks of interacting neurons. Over time, the interpretation of biology has become less literal, but the name stuck. At its heart lie a few key principles that can be found in most networks today:

- The alternation of linear and nonlinear processing units, often referred to as *layers*.

- The use of the chain rule (also known as *backpropagation*) for adjusting parameters in the entire network at once.

After initial rapid progress, research in neural networks languished from around 1995 until 2005. This was mainly due to two reasons. First, training a network is computationally very expensive. While random-access memory was plentiful at the end of the past century, computational power was scarce. Second, datasets were relatively small. In fact, Fisher's Iris dataset from 1936 was still a popular tool for testing the efficacy of algorithms. The MNIST dataset with its 60,000 handwritten digits was considered huge.

Given the scarcity of data and computation, strong statistical tools such as kernel methods, decision trees, and graphical models proved empirically superior in many applications. Moreover, unlike neural networks, they did not require weeks to train and provided predictable results with strong theoretical guarantees.

1.5 The Road to Deep Learning

Much of this changed with the availability of massive amounts of data, thanks to the World Wide Web, the advent of companies serving hundreds of millions of users online, a dissemination of low-cost, high-quality sensors, inexpensive data storage (Kryder's law), and cheap computation (Moore's law). In particular, the landscape of computation in deep learning was revolutionized by advances in GPUs that were originally engineered for computer gaming. Suddenly algorithms and models that seemed computationally infeasible were within reach. This is best illustrated in `tab_intro_decade`.

:Dataset vs. computer memory and computational power

Table 1.5.1: label:`tab_intro_decade`

Decade	Dataset	Memory	Floating point calculations per second
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (house prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (NVIDIA C2050)
2020	1 T (social network)	100 GB	1 PF (NVIDIA DGX-2)

Note that random-access memory has not kept pace with the growth in data. At the same time, increases in computational power have outpaced the growth in datasets. This means that statistical models need to become more memory efficient, and so they are free to spend more computer cycles optimizing parameters, thanks to the increased compute budget. Consequently, the sweet spot in machine learning and statistics moved from (generalized) linear models and kernel methods to deep neural networks. This is also one of the reasons why many of the mainstays of deep learning, such as multilayer perceptrons (McCulloch and Pitts, 1943), convolutional neural networks (LeCun *et al.*, 1998), long short-term memory (Hochreiter and Schmidhuber, 1997), and Q-Learning (Watkins and Dayan, 1992), were essentially "rediscovered" in the past decade, after lying comparatively dormant for considerable time.

The recent progress in statistical models, applications, and algorithms has sometimes been likened to the Cambrian explosion: a moment of rapid progress in the evolution of species. Indeed, the state of the art is not just a mere consequence of available resources applied to decades-old algorithms. Note that the list of ideas below barely scratches the surface of what has helped researchers achieve tremendous progress over the past decade.

- Novel methods for capacity control, such as *dropout* (Srivastava *et al.*, 2014), have helped to mitigate overfitting. Here, noise is injected (Bishop, 1995) throughout the neural network during training.

- *Attention mechanisms* solved a second problem that had plagued statistics for over a century: how to increase the memory and complexity of a system without increasing the number of learnable parameters. Researchers found an elegant solution by using what can only be viewed as a *learnable pointer structure* (Bahdanau *et al.*, 2014). Rather than having to remember an entire text sequence, e.g., for machine translation in a fixed-dimensional representation, all that needed to be stored was a pointer to the intermediate state of the translation process. This allowed for significantly increased accuracy for long sequences, since the model no longer needed to remember the entire sequence before commencing the generation of a new one.

- Built solely on attention mechanisms, the *Transformer* architecture (Vaswani *et al.*, 2017)

has demonstrated superior *scaling* behavior: it performs better with an increase in dataset size, model size, and amount of training compute (Kaplan *et al.*, 2020). This architecture has demonstrated compelling success in a wide range of areas, such as natural language processing (Brown *et al.*, 2020, Devlin *et al.*, 2018), computer vision (Dosovitskiy *et al.*, 2021, Liu *et al.*, 2021), speech recognition (Gulati *et al.*, 2020), reinforcement learning (Chen *et al.*, 2021), and graph neural networks (Dwivedi and Bresson, 2020). For example, a single Transformer pretrained on modalities as diverse as text, images, joint torques, and button presses can play Atari, caption images, chat, and control a robot (Reed *et al.*, 2022).

- Modeling probabilities of text sequences, *language models* can predict text given other text. Scaling up the data, model, and compute has unlocked a growing number of capabilities of language models to perform desired tasks via human-like text generation based on input text (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). For instance, aligning language models with human intent (Ouyang *et al.*, 2022), OpenAI's ChatGPT[30] allows users to interact with it in a conversational way to solve problems, such as code debugging and creative writing.

[30] 

- Multi-stage designs, e.g., via the memory networks (Sukhbaatar *et al.*, 2015) and the neural programmer-interpreter (Reed and De Freitas, 2015) permitted statistical modelers to describe iterative approaches to reasoning. These tools allow for an internal state of the deep neural network to be modified repeatedly, thus carrying out subsequent steps in a chain of reasoning, just as a processor can modify memory for a computation.

- A key development in *deep generative modeling* was the invention of *generative adversarial networks* (Goodfellow *et al.*, 2014). Traditionally, statistical methods for density estimation and generative models focused on finding proper probability distributions and (often approximate) algorithms for sampling from them. As a result, these algorithms were largely limited by the lack of flexibility inherent in the statistical models. The crucial innovation in generative adversarial networks was to replace the sampler by an arbitrary algorithm with differentiable parameters. These are then adjusted in such a way that the discriminator (effectively a two-sample test) cannot distinguish fake from real data. Through the ability to use arbitrary algorithms to generate data, density estimation was opened up to a wide variety of techniques. Examples of galloping zebras (Zhu *et al.*, 2017) and of fake celebrity faces (Karras *et al.*, 2017) are each testimony to this progress. Even amateur doodlers can produce photorealistic images just based on sketches describing the layout of a scene (Park *et al.*, 2019).

- Furthermore, while the diffusion process gradually adds random noise to data samples, *diffusion models* (Ho *et al.*, 2020, Sohl-Dickstein *et al.*, 2015) learn the denoising process to gradually construct data samples from random noise, reversing the diffusion process. They have started to replace generative adversarial networks in more recent deep generative models, such as in DALL-E 2 (Ramesh *et al.*, 2022) and Imagen (Saharia *et al.*, 2022) for creative art and image generation based on text descriptions.

- In many cases, a single GPU is insufficient for processing the large amounts of data

available for training. Over the past decade the ability to build parallel and distributed training algorithms has improved significantly. One of the key challenges in designing scalable algorithms is that the workhorse of deep learning optimization, stochastic gradient descent, relies on relatively small minibatches of data to be processed. At the same time, small batches limit the efficiency of GPUs. Hence, training on 1,024 GPUs with a minibatch size of, say, 32 images per batch amounts to an aggregate minibatch of about 32,000 images. Work, first by Li (2017) and subsequently by You *et al.* (2017) and Jia *et al.* (2018) pushed the size up to 64,000 observations, reducing training time for the ResNet-50 model on the ImageNet dataset to less than 7 minutes. By comparison, training times were initially of the order of days.

- The ability to parallelize computation has also contributed to progress in *reinforcement learning*. This has led to significant progress in computers achieving superhuman performance on tasks like Go, Atari games, Starcraft, and in physics simulations (e.g., using MuJoCo) where environment simulators are available. See, e.g., Silver *et al.* (2016) for a description of such achievements in AlphaGo. In a nutshell, reinforcement learning works best if plenty of (state, action, reward) tuples are available. Simulation provides such an avenue.

- Deep learning frameworks have played a crucial role in disseminating ideas. The first generation of open-source frameworks for neural network modeling consisted of Caffe [31], Torch [32], and Theano [33]. Many seminal papers were written using these tools. These have now been superseded by TensorFlow [34] (often used via its high-level API Keras [35]), CNTK [36], Caffe 2 [37], and Apache MXNet [38]. The third generation of frameworks consists of so-called *imperative* tools for deep learning, a trend that was arguably ignited by Chainer [39], which used a syntax similar to Python NumPy to describe models. This idea was adopted by both PyTorch [40], the Gluon API [41] of MXNet, and JAX [42].

The division of labor between system researchers building better tools and statistical modelers building better neural networks has greatly simplified things. For instance, training a linear logistic regression model used to be a nontrivial homework problem, worthy to give to new machine learning Ph.D. students at Carnegie Mellon University in 2014. By now, this task can be accomplished with under 10 lines of code, putting it firmly within the reach of any programmer.

1.6 Success Stories

Artificial intelligence has a long history of delivering results that would be difficult to accomplish otherwise. For instance, mail sorting systems using optical character recognition have been deployed since the 1990s. This is, after all, the source of the famous MNIST dataset of handwritten digits. The same applies to reading checks for bank deposits and scoring creditworthiness of applicants. Financial transactions are checked for fraud auto-

matically. This forms the backbone of many e-commerce payment systems, such as PayPal, Stripe, AliPay, WeChat, Apple, Visa, and MasterCard. Computer programs for chess have been competitive for decades. Machine learning feeds search, recommendation, personalization, and ranking on the Internet. In other words, machine learning is pervasive, albeit often hidden from sight.

It is only recently that AI has been in the limelight, mostly due to solutions to problems that were considered intractable previously and that are directly related to consumers. Many of such advances are attributed to deep learning.

- Intelligent assistants, such as Apple's Siri, Amazon's Alexa, and Google's assistant, are able to respond to spoken requests with a reasonable degree of accuracy. This includes menial jobs, like turning on light switches, and more complex tasks, such as arranging barber's appointments and offering phone support dialog. This is likely the most noticeable sign that AI is affecting our lives.

- A key ingredient in digital assistants is their ability to recognize speech accurately. The accuracy of such systems has gradually increased to the point of achieving parity with humans for certain applications (Xiong *et al.*, 2018).

- Object recognition has likewise come a long way. Identifying the object in a picture was a fairly challenging task in 2010. On the ImageNet benchmark researchers from NEC Labs and University of Illinois at Urbana-Champaign achieved a top-five error rate of 28% (Lin *et al.*, 2010). By 2017, this error rate was reduced to 2.25% (Hu *et al.*, 2018). Similarly, stunning results have been achieved for identifying birdsong and for diagnosing skin cancer.

- Prowess in games used to provide a measuring stick for human ability. Starting from TD-Gammon, a program for playing backgammon using temporal difference reinforcement learning, algorithmic and computational progress has led to algorithms for a wide range of applications. Compared with backgammon, chess has a much more complex state space and set of actions. DeepBlue beat Garry Kasparov using massive parallelism, special-purpose hardware and efficient search through the game tree (Campbell *et al.*, 2002). Go is more difficult still, due to its huge state space. AlphaGo reached human parity in 2015, using deep learning combined with Monte Carlo tree sampling (Silver *et al.*, 2016). The challenge in Poker was that the state space is large and only partially observed (we do not know the opponents' cards). Libratus exceeded human performance in Poker using efficiently structured strategies (Brown and Sandholm, 2017).

- Another indication of progress in AI is the advent of self-driving vehicles. While full autonomy is not yet within reach, excellent progress has been made in this direction, with companies such as Tesla, NVIDIA, and Waymo shipping products that enable partial autonomy. What makes full autonomy so challenging is that proper driving requires the ability to perceive, to reason and to incorporate rules into a system. At present, deep learning is used primarily in the visual aspect of these problems. The rest is heavily tuned by engineers.

This barely scratches the surface of significant applications of machine learning. For instance, robotics, logistics, computational biology, particle physics, and astronomy owe some of their most impressive recent advances at least in parts to machine learning, which is thus becoming a ubiquitous tool for engineers and scientists.

Frequently, questions about a coming AI apocalypse and the plausibility of a *singularity* have been raised in non-technical articles. The fear is that somehow machine learning systems will become sentient and make decisions, independently of their programmers, that directly impact the lives of humans. To some extent, AI already affects the livelihood of humans in direct ways: creditworthiness is assessed automatically, autopilots mostly navigate vehicles, decisions about whether to grant bail use statistical data as input. More frivolously, we can ask Alexa to switch on the coffee machine.

Fortunately, we are far from a sentient AI system that could deliberately manipulate its human creators. First, AI systems are engineered, trained, and deployed in a specific, goal-oriented manner. While their behavior might give the illusion of general intelligence, it is a combination of rules, heuristics and statistical models that underlie the design. Second, at present, there are simply no tools for *artificial general intelligence* that are able to improve themselves, reason about themselves, and that are able to modify, extend, and improve their own architecture while trying to solve general tasks.

A much more pressing concern is how AI is being used in our daily lives. It is likely that many routine tasks, currently fulfilled by humans, can and will be automated. Farm robots will likely reduce the costs for organic farmers but they will also automate harvesting operations. This phase of the industrial revolution may have profound consequences for large swaths of society, since menial jobs provide much employment in many countries. Furthermore, statistical models, when applied without care, can lead to racial, gender, or age bias and raise reasonable concerns about procedural fairness if automated to drive consequential decisions. It is important to ensure that these algorithms are used with care. With what we know today, this strikes us as a much more pressing concern than the potential of malevolent superintelligence for destroying humanity.

1.7 The Essence of Deep Learning

Thus far, we have talked in broad terms about machine learning. Deep learning is the subset of machine learning concerned with models based on many-layered neural networks. It is *deep* in precisely the sense that its models learn many *layers* of transformations. While this might sound narrow, deep learning has given rise to a dizzying array of models, techniques, problem formulations, and applications. Many intuitions have been developed to explain the benefits of depth. Arguably, all machine learning has many layers of computation, the first consisting of feature processing steps. What differentiates deep learning is that the operations learned at each of the many layers of representations are learned jointly from data.

The problems that we have discussed so far, such as learning from the raw audio signal, the raw pixel values of images, or mapping between sentences of arbitrary lengths and their counterparts in foreign languages, are those where deep learning excels and traditional methods falter. It turns out that these many-layered models are capable of addressing low-level perceptual data in a way that previous tools could not. Arguably the most significant commonality in deep learning methods is *end-to-end training*. That is, rather than assembling a system based on components that are individually tuned, one builds the system and then tunes their performance jointly. For instance, in computer vision scientists used to separate the process of *feature engineering* from the process of building machine learning models. The Canny edge detector (Canny, 1987) and Lowe's SIFT feature extractor (Lowe, 2004) reigned supreme for over a decade as algorithms for mapping images into feature vectors. In bygone days, the crucial part of applying machine learning to these problems consisted of coming up with manually-engineered ways of transforming the data into some form amenable to shallow models. Unfortunately, there is only so much that humans can accomplish by ingenuity in comparison with a consistent evaluation over millions of choices carried out automatically by an algorithm. When deep learning took over, these feature extractors were replaced by automatically tuned filters that yielded superior accuracy.

Thus, one key advantage of deep learning is that it replaces not only the shallow models at the end of traditional learning pipelines, but also the labor-intensive process of feature engineering. Moreover, by replacing much of the domain-specific preprocessing, deep learning has eliminated many of the boundaries that previously separated computer vision, speech recognition, natural language processing, medical informatics, and other application areas, thereby offering a unified set of tools for tackling diverse problems.

Beyond end-to-end training, we are experiencing a transition from parametric statistical descriptions to fully nonparametric models. When data is scarce, one needs to rely on simplifying assumptions about reality in order to obtain useful models. When data is abundant, these can be replaced by nonparametric models that better fit the data. To some extent, this mirrors the progress that physics experienced in the middle of the previous century with the availability of computers. Rather than solving by hand parametric approximations of how electrons behave, one can now resort to numerical simulations of the associated partial differential equations. This has led to much more accurate models, albeit often at the expense of interpretation.

Another difference from previous work is the acceptance of suboptimal solutions, dealing with nonconvex nonlinear optimization problems, and the willingness to try things before proving them. This new-found empiricism in dealing with statistical problems, combined with a rapid influx of talent has led to rapid progress in the development of practical algorithms, albeit in many cases at the expense of modifying and re-inventing tools that existed for decades.

In the end, the deep learning community prides itself on sharing tools across academic and corporate boundaries, releasing many excellent libraries, statistical models, and trained networks as open source. It is in this spirit that the notebooks forming this book are freely available for distribution and use. We have worked hard to lower the barriers of access for

anyone wishing to learn about deep learning and we hope that our readers will benefit from this.

1.8 Summary

Machine learning studies how computer systems can leverage experience (often data) to improve performance at specific tasks. It combines ideas from statistics, data mining, and optimization. Often, it is used as a means of implementing AI solutions. As a class of machine learning, representational learning focuses on how to automatically find the appropriate way to represent data. Considered as multi-level representation learning through learning many layers of transformations, deep learning replaces not only the shallow models at the end of traditional machine learning pipelines, but also the labor-intensive process of feature engineering. Much of the recent progress in deep learning has been triggered by an abundance of data arising from cheap sensors and Internet-scale applications, and by significant progress in computation, mostly through GPUs. Furthermore, the availability of efficient deep learning frameworks has made design and implementation of whole system optimization significantly easier, and this is a key component in obtaining high performance.

1.9 Exercises

1. Which parts of code that you are currently writing could be "learned", i.e., improved by learning and automatically determining design choices that are made in your code? Does your code include heuristic design choices? What data might you need to learn the desired behavior?
2. Which problems that you encounter have many examples for their solution, yet no specific way for automating them? These may be prime candidates for using deep learning.
3. Describe the relationships between algorithms, data, and computation. How do characteristics of the data and the current available computational resources influence the appropriateness of various algorithms?
4. Name some settings where end-to-end training is not currently the default approach but where it might be useful.

Discussions[43].

43 

2 Preliminaries

To prepare for your dive into deep learning, you will need a few survival skills: (i) techniques for storing and manipulating data; (ii) libraries for ingesting and preprocessing data from a variety of sources; (iii) knowledge of the basic linear algebraic operations that we apply to high-dimensional data elements; (iv) just enough calculus to determine which direction to adjust each parameter in order to decrease the loss function; (v) the ability to automatically compute derivatives so that you can forget much of the calculus you just learned; (vi) some basic fluency in probability, our primary language for reasoning under uncertainty; and (vii) some aptitude for finding answers in the official documentation when you get stuck.

In short, this chapter provides a rapid introduction to the basics that you will need to follow *most* of the technical content in this book.

2.1 Data Manipulation

In order to get anything done, we need some way to store and manipulate data. Generally, there are two important things we need to do with data: (i) acquire them; and (ii) process them once they are inside the computer. There is no point in acquiring data without some way to store it, so to start, let's get our hands dirty with n-dimensional arrays, which we also call *tensors*. If you already know the NumPy scientific computing package, this will be a breeze. For all modern deep learning frameworks, the *tensor class* (`ndarray` in MXNet, `Tensor` in PyTorch and TensorFlow) resembles NumPy's `ndarray`, with a few killer features added. First, the tensor class supports automatic differentiation. Second, it leverages GPUs to accelerate numerical computation, whereas NumPy only runs on CPUs. These properties make neural networks both easy to code and fast to run.

2.1.1 Getting Started

To start, we import the PyTorch library. Note that the package name is `torch`.

```
import torch
```

A tensor represents a (possibly multidimensional) array of numerical values. In the one-dimensional case, i.e., when only one axis is needed for the data, a tensor is called a *vector*.

With two axes, a tensor is called a *matrix*. With $k > 2$ axes, we drop the specialized names and just refer to the object as a k^{th}*-order tensor*.

PyTorch provides a variety of functions for creating new tensors prepopulated with values. For example, by invoking `arange(n)`, we can create a vector of evenly spaced values, starting at 0 (included) and ending at `n` (not included). By default, the interval size is 1. Unless otherwise specified, new tensors are stored in main memory and designated for CPU-based computation.

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

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

Each of these values is called an *element* of the tensor. The tensor `x` contains 12 elements. We can inspect the total number of elements in a tensor via its `numel` method.

```
x.numel()
```

```
12
```

We can access a tensor's *shape* (the length along each axis) by inspecting its `shape` attribute. Because we are dealing with a vector here, the `shape` contains just a single element and is identical to the size.

```
x.shape
```

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

We can change the shape of a tensor without altering its size or values, by invoking `reshape`. For example, we can transform our vector `x` whose shape is (12,) to a matrix `X` with shape (3, 4). This new tensor retains all elements but reconfigures them into a matrix. Notice that the elements of our vector are laid out one row at a time and thus `x[3] == X[0, 3]`.

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

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

Note that specifying every shape component to `reshape` is redundant. Because we already know our tensor's size, we can work out one component of the shape given the rest. For example, given a tensor of size n and target shape (h, w), we know that $w = n/h$. To

automatically infer one component of the shape, we can place a `-1` for the shape component that should be inferred automatically. In our case, instead of calling `x.reshape(3, 4)`, we could have equivalently called `x.reshape(-1, 4)` or `x.reshape(3, -1)`.

Practitioners often need to work with tensors initialized to contain all 0s or 1s. We can construct a tensor with all elements set to 0 and a shape of (2, 3, 4) via the `zeros` function.

```
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.]]])
```

Similarly, we can create a tensor with all 1s by invoking `ones`.

```
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.]]])
```

We often wish to sample each element randomly (and independently) from a given probability distribution. For example, the parameters of neural networks are often initialized randomly. The following snippet creates a tensor with elements drawn from a standard Gaussian (normal) distribution with mean 0 and standard deviation 1.

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

```
tensor([[ 0.1351, -0.9099, -0.2028,  2.1937],
        [-0.3200, -0.7545,  0.8086, -1.8730],
        [ 0.3929,  0.4931,  0.9114, -0.7072]])
```

Finally, we can construct tensors by supplying the exact values for each element by supplying (possibly nested) Python list(s) containing numerical literals. Here, we construct a matrix with a list of lists, where the outermost list corresponds to axis 0, and the inner list corresponds to axis 1.

```
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]])
```

2.1.2 Indexing and Slicing

As with Python lists, we can access tensor elements by indexing (starting with 0). To access an element based on its position relative to the end of the list, we can use negative indexing. Finally, we can access whole ranges of indices via slicing (e.g., `X[start:stop]`), where the returned value includes the first index (`start`) *but not the last* (`stop`). Finally, when only one index (or slice) is specified for a k^{th}-order tensor, it is applied along axis 0. Thus, in the following code, `[-1]` selects the last row and `[1:3]` selects the second and third rows.

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

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

Beyond reading them, we can also *write* elements of a matrix by specifying indices.

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

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

If we want to assign multiple elements the same value, we apply the indexing on the left-hand side of the assignment operation. For instance, `[:2, :]` accesses the first and second rows, where `:` takes all the elements along axis 1 (column). While we discussed indexing for matrices, this also works for vectors and for tensors of more than two dimensions.

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

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

2.1.3 Operations

Now that we know how to construct tensors and how to read from and write to their elements, we can begin to manipulate them with various mathematical operations. Among the most useful of these are the *elementwise* operations. These apply a standard scalar operation to each element of a tensor. For functions that take two tensors as inputs, elementwise operations apply some standard binary operator on each pair of corresponding elements. We can create an elementwise function from any function that maps from a scalar to a scalar.

In mathematical notation, we denote such *unary* scalar operators (taking one input) by the signature $f : \mathbb{R} \rightarrow \mathbb{R}$. This just means that the function maps from any real number onto some other real number. Most standard operators, including unary ones like e^x, can be applied elementwise.

```
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])
```

Likewise, we denote *binary* scalar operators, which map pairs of real numbers to a (single) real number via the signature $f : \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$. Given any two vectors $\mathbf{u}$ and $\mathbf{v}$ *of the same shape*, and a binary operator f, we can produce a vector $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ by setting $c_i \leftarrow f(u_i, v_i)$ for all i, where c_i, u_i, and v_i are the i^{th} elements of vectors $\mathbf{c}, \mathbf{u}$, and $\mathbf{v}$. Here, we produced the vector-valued $F : \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$ by *lifting* the scalar function to an elementwise vector operation. The common standard arithmetic operators for addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`**`) have all been *lifted* to elementwise operations for identically-shaped tensors of arbitrary shape.

```
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.]))
```

In addition to elementwise computations, we can also perform linear algebraic operations, such as dot products and matrix multiplications. We will elaborate on these in Section 2.3.

We can also *concatenate* multiple tensors, stacking them end-to-end to form a larger one. We just need to provide a list of tensors and tell the system along which axis to concatenate. The example below shows what happens when we concatenate two matrices along rows

(axis 0) instead of columns (axis 1). We can see that the first output's axis-0 length (6) is the sum of the two input tensors' axis-0 lengths (3 + 3); while the second output's axis-1 length (8) is the sum of the two input tensors' axis-1 lengths (4 + 4).

```
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.]]))
```

Sometimes, we want to construct a binary tensor via *logical statements*. Take `X == Y` as an example. For each position `i, j`, if `X[i, j]` and `Y[i, j]` are equal, then the corresponding entry in the result takes value 1, otherwise it takes value `0`.

```
X == Y
```

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

Summing all the elements in the tensor yields a tensor with only one element.

```
X.sum()
```

```
tensor(66.)
```

2.1.4 Broadcasting

By now, you know how to perform elementwise binary operations on two tensors of the same shape. Under certain conditions, even when shapes differ, we can still perform elementwise binary operations by invoking the *broadcasting mechanism*. Broadcasting works according to the following two-step procedure: (i) expand one or both arrays by copying elements along axes with length 1 so that after this transformation, the two tensors have the same shape; (ii) perform an elementwise operation on the resulting arrays.

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

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

Since `a` and `b` are 3×1 and 1×2 matrices, respectively, their shapes do not match up. Broadcasting produces a larger 3×2 matrix by replicating matrix `a` along the columns and matrix `b` along the rows before adding them elementwise.

```
a + b
```

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

2.1.5 Saving Memory

Running operations can cause new memory to be allocated to host results. For example, if we write `Y = X + Y`, we dereference the tensor that `Y` used to point to and instead point `Y` at the newly allocated memory. We can demonstrate this issue with Python's `id()` function, which gives us the exact address of the referenced object in memory. Note that after we run `Y = Y + X`, `id(Y)` points to a different location. That is because Python first evaluates `Y + X`, allocating new memory for the result and then points `Y` to this new location in memory.

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

```
False
```

This might be undesirable for two reasons. First, we do not want to run around allocating memory unnecessarily all the time. In machine learning, we often have hundreds of megabytes of parameters and update all of them multiple times per second. Whenever possible, we want to perform these updates *in place*. Second, we might point at the same parameters from multiple variables. If we do not update in place, we must be careful to update all of these references, lest we spring a memory leak or inadvertently refer to stale parameters.

Fortunately, performing in-place operations is easy. We can assign the result of an operation to a previously allocated array `Y` by using slice notation: `Y[:] = <expression>`. To illustrate this concept, we overwrite the values of tensor `Z`, after initializing it, using `zeros_like`, to have the same shape as `Y`.

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

```
id(Z): 140381179266448
id(Z): 140381179266448
```

If the value of `X` is not reused in subsequent computations, we can also use `X[:] = X + Y` or `X += Y` to reduce the memory overhead of the operation.

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

```
True
```

2.1.6 Conversion to Other Python Objects

Converting to a NumPy tensor (`ndarray`), or vice versa, is easy. The torch tensor and NumPy array will share their underlying memory, and changing one through an in-place operation will also change the other.

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

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

To convert a size-1 tensor to a Python scalar, we can invoke the `item` function or Python's built-in functions.

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

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

2.1.7 Summary

The tensor class is the main interface for storing and manipulating data in deep learning libraries. Tensors provide a variety of functionalities including construction routines; indexing and slicing; basic mathematics operations; broadcasting; memory-efficient assignment; and conversion to and from other Python objects.

2.1.8 Exercises

1. Run the code in this section. Change the conditional statement `X == Y` to `X < Y` or `X > Y`, and then see what kind of tensor you can get.

2. Replace the two tensors that operate by element in the broadcasting mechanism with other shapes, e.g., 3-dimensional tensors. Is the result the same as expected?

44 Discussions[44].

2.2 Data Preprocessing

So far, we have been working with synthetic data that arrived in ready-made tensors. However, to apply deep learning in the wild we must extract messy data stored in arbitrary
45 formats, and preprocess it to suit our needs. Fortunately, the *pandas* library[45] can do much of the heavy lifting. This section, while no substitute for a proper *pandas* tutorial[46], will give you a crash course on some of the most common routines.

46

2.2.1 Reading the Dataset

Comma-separated values (CSV) files are ubiquitous for the storing of tabular (spreadsheet-like) data. In them, each line corresponds to one record and consists of several (comma-separated) fields, e.g., "Albert Einstein,March 14 1879,Ulm,Federal polytechnic school,field of gravitational physics". To demonstrate how to load CSV files with `pandas`, we create a CSV file below `../data/house_tiny.csv`. This file represents a dataset of homes, where each row corresponds to a distinct home and the columns correspond to the number of rooms (`NumRooms`), the roof type (`RoofType`), and the price (`Price`).

```
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''')
```

Now let's import `pandas` and load the dataset with `read_csv`.

```
import pandas as pd

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

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

2.2.2 Data Preparation

In supervised learning, we train models to predict a designated *target* value, given some set of *input* values. Our first step in processing the dataset is to separate out columns corresponding to input versus target values. We can select columns either by name or via integer-location based indexing (`iloc`).

You might have noticed that `pandas` replaced all CSV entries with value `NA` with a special `NaN` (*not a number*) value. This can also happen whenever an entry is empty, e.g., "3,,,270000". These are called *missing values* and they are the "bed bugs" of data science, a persistent menace that you will confront throughout your career. Depending upon the context, missing values might be handled either via *imputation* or *deletion*. Imputation replaces missing values with estimates of their values while deletion simply discards either those rows or those columns that contain missing values.

Here are some common imputation heuristics. For categorical input fields, we can treat `NaN` as a category. Since the `RoofType` column takes values `Slate` and `NaN`, `pandas` can convert this column into two columns `RoofType_Slate` and `RoofType_nan`. A row whose roof type is `Slate` will set values of `RoofType_Slate` and `RoofType_nan` to 1 and 0, respectively. The converse holds for a row with a missing `RoofType` value.

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

For missing numerical values, one common heuristic is to replace the `NaN` entries with the mean value of the corresponding column.

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

```
   NumRooms  RoofType_Slate  RoofType_nan
0       3.0           False          True
1       2.0           False          True
```

(continues on next page)

(continued from previous page)

```
2       4.0            True         False
3       3.0           False          True
```

2.2.3 Conversion to the Tensor Format

Now that all the entries in `inputs` and `targets` are numerical, we can load them into a tensor (recall Section 2.1).

```
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))
```

2.2.4 Discussion

You now know how to partition data columns, impute missing variables, and load `pandas` data into tensors. In Section 5.7, you will pick up some more data processing skills. While this crash course kept things simple, data processing can get hairy. For example, rather than arriving in a single CSV file, our dataset might be spread across multiple files extracted from a relational database. For instance, in an e-commerce application, customer addresses might live in one table and purchase data in another. Moreover, practitioners face myriad data types beyond categorical and numeric, for example, text strings, images, audio data, and point clouds. Oftentimes, advanced tools and efficient algorithms are required in order to prevent data processing from becoming the biggest bottleneck in the machine learning pipeline. These problems will arise when we get to computer vision and natural language processing. Finally, we must pay attention to data quality. Real-world datasets are often plagued by outliers, faulty measurements from sensors, and recording errors, which must be addressed before feeding the data into any model. Data visualization tools such as seaborn[47], Bokeh[48], or matplotlib[49] can help you to manually inspect the data and develop intuitions about the type of problems you may need to address.

2.2.5 Exercises

1. Try loading datasets, e.g., Abalone from the UCI Machine Learning Repository[50] and inspect their properties. What fraction of them has missing values? What fraction of the variables is numerical, categorical, or text?
2. Try indexing and selecting data columns by name rather than by column number. The pandas documentation on indexing[51] has further details on how to do this.

3. How large a dataset do you think you could load this way? What might be the limitations? Hint: consider the time to read the data, representation, processing, and memory footprint. Try this out on your laptop. What happens if you try it out on a server?

4. How would you deal with data that has a very large number of categories? What if the category labels are all unique? Should you include the latter?

5. What alternatives to pandas can you think of? How about loading NumPy tensors from a file[52]? Check out Pillow[53], the Python Imaging Library.

52 

Discussions[54].

53 

2.3 Linear Algebra

54 

By now, we can load datasets into tensors and manipulate these tensors with basic mathematical operations. To start building sophisticated models, we will also need a few tools from linear algebra. This section offers a gentle introduction to the most essential concepts, starting from scalar arithmetic and ramping up to matrix multiplication.

```
import torch
```

2.3.1 Scalars

Most everyday mathematics consists of manipulating numbers one at a time. Formally, we call these values *scalars*. For example, the temperature in Palo Alto is a balmy 72 degrees Fahrenheit. If you wanted to convert the temperature to Celsius you would evaluate the expression $c = \frac{5}{9}(f - 32)$, setting f to 72. In this equation, the values 5, 9, and 32 are constant scalars. The variables c and f in general represent unknown scalars.

We denote scalars by ordinary lower-cased letters (e.g., x, y, and z) and the space of all (continuous) *real-valued* scalars by $\mathbb{R}$. For expedience, we will skip past rigorous definitions of *spaces*: just remember that the expression $x \in \mathbb{R}$ is a formal way to say that x is a real-valued scalar. The symbol $\in$ (pronounced "in") denotes membership in a set. For example, $x, y \in \{0, 1\}$ indicates that x and y are variables that can only take values 0 or 1.

Scalars are implemented as tensors that contain only one element. Below, we assign two scalars and perform the familiar addition, multiplication, division, and exponentiation operations.

```
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.))
```

2.3.2 Vectors

For current purposes, you can think of a vector as a fixed-length array of scalars. As with their code counterparts, we call these scalars the *elements* of the vector (synonyms include *entries* and *components*). When vectors represent examples from real-world datasets, their values hold some real-world significance. For example, if we were training a model to predict the risk of a loan defaulting, we might associate each applicant with a vector whose components correspond to quantities like their income, length of employment, or number of previous defaults. If we were studying the risk of heart attack, each vector might represent a patient and its components might correspond to their most recent vital signs, cholesterol levels, minutes of exercise per day, etc. We denote vectors by bold lowercase letters, (e.g., $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$).

Vectors are implemented as 1^{st}-order tensors. In general, such tensors can have arbitrary lengths, subject to memory limitations. Caution: in Python, as in most programming languages, vector indices start at 0, also known as *zero-based indexing*, whereas in linear algebra subscripts begin at 1 (one-based indexing).

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

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

We can refer to an element of a vector by using a subscript. For example, x_2 denotes the second element of $\mathbf{x}$. Since x_2 is a scalar, we do not bold it. By default, we visualize vectors by stacking their elements vertically.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \tag{2.3.1}$$

Here $x_1, \ldots, x_n$ are elements of the vector. Later on, we will distinguish between such *column vectors* and *row vectors* whose elements are stacked horizontally. Recall that we access a tensor's elements via indexing.

```
x[2]
```

```
tensor(2)
```

To indicate that a vector contains n elements, we write $\mathbf{x} \in \mathbb{R}^n$. Formally, we call n the *dimensionality* of the vector. In code, this corresponds to the tensor's length, accessible via Python's built-in `len` function.

```
len(x)
```

```
3
```

We can also access the length via the shape attribute. The shape is a tuple that indicates a tensor's length along each axis. Tensors with just one axis have shapes with just one element.

```
x.shape
```

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

Oftentimes, the word "dimension" gets overloaded to mean both the number of axes and the length along a particular axis. To avoid this confusion, we use *order* to refer to the number of axes and *dimensionality* exclusively to refer to the number of components.

2.3.3 Matrices

Just as scalars are 0^{th}-order tensors and vectors are 1^{st}-order tensors, matrices are 2^{nd}-order tensors. We denote matrices by bold capital letters (e.g., $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$), and represent them in code by tensors with two axes. The expression $\mathbf{A} \in \mathbb{R}^{m \times n}$ indicates that a matrix $\mathbf{A}$ contains $m \times n$ real-valued scalars, arranged as m rows and n columns. When $m = n$, we say that a matrix is *square*. Visually, we can illustrate any matrix as a table. To refer to an individual element, we subscript both the row and column indices, e.g., a_{ij} is the value that belongs to $\mathbf{A}$'s i^{th} row and j^{th} column:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{2.3.2}$$

In code, we represent a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ by a 2^{nd}-order tensor with shape (m, n). We can convert any appropriately sized $m \times n$ tensor into an $m \times n$ matrix by passing the desired shape to reshape:

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

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

Sometimes we want to flip the axes. When we exchange a matrix's rows and columns, the result is called its *transpose*. Formally, we signify a matrix $\mathbf{A}$'s transpose by $\mathbf{A}^\top$ and if

$\mathbf{B} = \mathbf{A}^\top$, then $b_{ij} = a_{ji}$ for all i and j. Thus, the transpose of an $m \times n$ matrix is an $n \times m$ matrix:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}. \tag{2.3.3}$$

In code, we can access any matrix's transpose as follows:

```
A.T
```

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

Symmetric matrices are the subset of square matrices that are equal to their own transposes: $\mathbf{A} = \mathbf{A}^\top$. The following matrix is symmetric:

```
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]])
```

Matrices are useful for representing datasets. Typically, rows correspond to individual records and columns correspond to distinct attributes.

2.3.4 Tensors

While you can go far in your machine learning journey with only scalars, vectors, and matrices, eventually you may need to work with higher-order tensors. Tensors give us a generic way of describing extensions to n^{th}-order arrays. We call software objects of the *tensor class* "tensors" precisely because they too can have arbitrary numbers of axes. While it may be confusing to use the word *tensor* for both the mathematical object and its realization in code, our meaning should usually be clear from context. We denote general tensors by capital letters with a special font face (e.g., X, Y, and Z) and their indexing mechanism (e.g., x_{ijk} and $[\mathsf{X}]_{1,2i-1,3}$) follows naturally from that of matrices.

Tensors will become more important when we start working with images. Each image arrives as a 3^{rd}-order tensor with axes corresponding to the height, width, and *channel*. At each spatial location, the intensities of each color (red, green, and blue) are stacked along the channel. Furthermore, a collection of images is represented in code by a 4^{th}-order tensor, where distinct images are indexed along the first axis. Higher-order tensors are constructed, as were vectors and matrices, by growing the number of shape components.

```
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]]])
```

2.3.5 Basic Properties of Tensor Arithmetic

Scalars, vectors, matrices, and higher-order tensors all have some handy properties. For example, elementwise operations produce outputs that have the same shape as their operands.

```
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.]]))
```

The elementwise product of two matrices is called their *Hadamard product* (denoted $\odot$). We can spell out the entries of the Hadamard product of two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$:

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \tag{2.3.4}$$

```
A * B
```

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

Adding or multiplying a scalar and a tensor produces a result with the same shape as the original tensor. Here, each element of the tensor is added to (or multiplied by) the scalar.

```
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]))
```

2.3.6 Reduction

Often, we wish to calculate the sum of a tensor's elements. To express the sum of the elements in a vector $\mathbf{x}$ of length n, we write $\sum_{i=1}^{n} x_i$. There is a simple function for it:

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

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

To express sums over the elements of tensors of arbitrary shape, we simply sum over all its axes. For example, the sum of the elements of an $m \times n$ matrix $\mathbf{A}$ could be written $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$.

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

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

By default, invoking the sum function *reduces* a tensor along all of its axes, eventually producing a scalar. Our libraries also allow us to specify the axes along which the tensor should be reduced. To sum over all elements along the rows (axis 0), we specify `axis=0` in `sum`. Since the input matrix reduces along axis 0 to generate the output vector, this axis is missing from the shape of the output.

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

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

Specifying `axis=1` will reduce the column dimension (axis 1) by summing up elements of all the columns.

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

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

Reducing a matrix along both rows and columns via summation is equivalent to summing up all the elements of the matrix.

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

```
tensor(True)
```

A related quantity is the *mean*, also called the *average*. We calculate the mean by dividing the sum by the total number of elements. Because computing the mean is so common, it gets a dedicated library function that works analogously to `sum`.

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

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

Likewise, the function for calculating the mean can also reduce a tensor along specific axes.

```
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]))
```

2.3.7 Non-Reduction Sum

Sometimes it can be useful to keep the number of axes unchanged when invoking the function for calculating the sum or mean. This matters when we want to use the broadcast mechanism.

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

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

For instance, since `sum_A` keeps its two axes after summing each row, we can divide `A` by `sum_A` with broadcasting to create a matrix where each row sums up to 1.

```
A / sum_A
```

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

If we want to calculate the cumulative sum of elements of `A` along some axis, say `axis=0` (row by row), we can call the `cumsum` function. By design, this function does not reduce the input tensor along any axis.

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

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

2.3.8 Dot Products

So far, we have only performed elementwise operations, sums, and averages. And if this was all we could do, linear algebra would not deserve its own section. Fortunately, this is where things get more interesting. One of the most fundamental operations is the dot product. Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their *dot product* $\mathbf{x}^\top \mathbf{y}$ (also known as *inner product*, $\langle \mathbf{x}, \mathbf{y} \rangle$) is a sum over the products of the elements at the same position: $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$.

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

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

Equivalently, we can calculate the dot product of two vectors by performing an elementwise multiplication followed by a sum:

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

```
tensor(3.)
```

Dot products are useful in a wide range of contexts. For example, given some set of values, denoted by a vector $\mathbf{x} \in \mathbb{R}^n$, and a set of weights, denoted by $\mathbf{w} \in \mathbb{R}^n$, the weighted sum of the values in $\mathbf{x}$ according to the weights $\mathbf{w}$ could be expressed as the dot product $\mathbf{x}^\top \mathbf{w}$. When the weights are nonnegative and sum to 1, i.e., $\left(\sum_{i=1}^{n} w_i = 1\right)$, the dot product expresses a *weighted average*. After normalizing two vectors to have unit length, the dot products express the cosine of the angle between them. Later in this section, we will formally introduce this notion of *length*.

2.3.9 Matrix–Vector Products

Now that we know how to calculate dot products, we can begin to understand the *product* between an $m \times n$ matrix $\mathbf{A}$ and an n-dimensional vector $\mathbf{x}$. To start off, we visualize our

matrix in terms of its row vectors

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \tag{2.3.5}$$

where each $\mathbf{a}_i^\top \in \mathbb{R}^n$ is a row vector representing the i^{th} row of the matrix $\mathbf{A}$.

The matrix–vector product $\mathbf{Ax}$ is simply a column vector of length m, whose i^{th} element is the dot product $\mathbf{a}_i^\top \mathbf{x}$:

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}. \tag{2.3.6}$$

We can think of multiplication with a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ as a transformation that projects vectors from $\mathbb{R}^n$ to $\mathbb{R}^m$. These transformations are remarkably useful. For example, we can represent rotations as multiplications by certain square matrices. Matrix–vector products also describe the key calculation involved in computing the outputs of each layer in a neural network given the outputs from the previous layer.

To express a matrix–vector product in code, we use the `mv` function. Note that the column dimension of `A` (its length along axis 1) must be the same as the dimension of `x` (its length). Python has a convenience operator `@` that can execute both matrix–vector and matrix–matrix products (depending on its arguments). Thus we can write `A@x`.

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

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

2.3.10 Matrix–Matrix Multiplication

Once you have gotten the hang of dot products and matrix–vector products, then *matrix–matrix multiplication* should be straightforward.

Say that we have two matrices $\mathbf{A} \in \mathbb{R}^{n \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times m}$:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}. \tag{2.3.7}$$

Let $\mathbf{a}_i^\top \in \mathbb{R}^k$ denote the row vector representing the i^{th} row of the matrix $\mathbf{A}$ and let $\mathbf{b}_j \in \mathbb{R}^k$

denote the column vector from the j^{th} column of the matrix $\mathbf{B}$:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix}. \tag{2.3.8}$$

To form the matrix product $\mathbf{C} \in \mathbb{R}^{n \times m}$, we simply compute each element c_{ij} as the dot product between the i^{th} row of $\mathbf{A}$ and the j^{th} column of $\mathbf{B}$, i.e., $\mathbf{a}_i^\top \mathbf{b}_j$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}. \tag{2.3.9}$$

We can think of the matrix–matrix multiplication $\mathbf{AB}$ as performing m matrix–vector products or $m \times n$ dot products and stitching the results together to form an $n \times m$ matrix. In the following snippet, we perform matrix multiplication on `A` and `B`. Here, `A` is a matrix with two rows and three columns, and `B` is a matrix with three rows and four columns. After multiplication, we obtain a matrix with two rows and four columns.

```
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.]]))
```

The term *matrix–matrix multiplication* is often simplified to *matrix multiplication*, and should not be confused with the Hadamard product.

2.3.11 Norms

Some of the most useful operators in linear algebra are *norms*. Informally, the norm of a vector tells us how *big* it is. For instance, the ℓ_2 norm measures the (Euclidean) length of a vector. Here, we are employing a notion of *size* that concerns the magnitude of a vector's components (not its dimensionality).

A norm is a function $\|\cdot\|$ that maps a vector to a scalar and satisfies the following three properties:

1. Given any vector $\mathbf{x}$, if we scale (all elements of) the vector by a scalar $\alpha \in \mathbb{R}$, its norm scales accordingly:

$$\|\alpha \mathbf{x}\| = |\alpha| \|\mathbf{x}\|. \tag{2.3.10}$$

2. For any vectors $\mathbf{x}$ and $\mathbf{y}$: norms satisfy the triangle inequality:

$$\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|. \tag{2.3.11}$$

3. The norm of a vector is nonnegative and it only vanishes if the vector is zero:

$$\|\mathbf{x}\| > 0 \text{ for all } \mathbf{x} \neq 0. \tag{2.3.12}$$

Many functions are valid norms and different norms encode different notions of size. The Euclidean norm that we all learned in elementary school geometry when calculating the hypotenuse of a right triangle is the square root of the sum of squares of a vector's elements. Formally, this is called the ℓ_2 *norm* and expressed as

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}. \tag{2.3.13}$$

The method `norm` calculates the ℓ_2 norm.

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

```
tensor(5.)
```

The ℓ_1 norm is also common and the associated measure is called the Manhattan distance. By definition, the ℓ_1 norm sums the absolute values of a vector's elements:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} \left|x_i\right|. \tag{2.3.14}$$

Compared to the ℓ_2 norm, it is less sensitive to outliers. To compute the ℓ_1 norm, we compose the absolute value with the sum operation.

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

```
tensor(7.)
```

Both the ℓ_2 and ℓ_1 norms are special cases of the more general ℓ_p *norms*:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} \left|x_i\right|^p\right)^{1/p}. \tag{2.3.15}$$

In the case of matrices, matters are more complicated. After all, matrices can be viewed both as collections of individual entries *and* as objects that operate on vectors and transform them into other vectors. For instance, we can ask by how much longer the matrix–vector product $\mathbf{X}\mathbf{v}$ could be relative to $\mathbf{v}$. This line of thought leads to what is called the *spectral*

norm. For now, we introduce the *Frobenius norm*, which is much easier to compute and defined as the square root of the sum of the squares of a matrix's elements:

$$\|\mathbf{X}\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}^2}. \tag{2.3.16}$$

The Frobenius norm behaves as if it were an ℓ_2 norm of a matrix-shaped vector. Invoking the following function will calculate the Frobenius norm of a matrix.

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

```
tensor(6.)
```

While we do not want to get too far ahead of ourselves, we already can plant some intuition about why these concepts are useful. In deep learning, we are often trying to solve optimization problems: *maximize* the probability assigned to observed data; *maximize* the revenue associated with a recommender model; *minimize* the distance between predictions and the ground truth observations; *minimize* the distance between representations of photos of the same person while *maximizing* the distance between representations of photos of different people. These distances, which constitute the objectives of deep learning algorithms, are often expressed as norms.

2.3.12 Discussion

In this section, we have reviewed all the linear algebra that you will need to understand a significant chunk of modern deep learning. There is a lot more to linear algebra, though, and much of it is useful for machine learning. For example, matrices can be decomposed into factors, and these decompositions can reveal low-dimensional structure in real-world datasets. There are entire subfields of machine learning that focus on using matrix decompositions and their generalizations to high-order tensors to discover structure in datasets and solve prediction problems. But this book focuses on deep learning. And we believe you will be more inclined to learn more mathematics once you have gotten your hands dirty applying machine learning to real datasets. So while we reserve the right to introduce more mathematics later on, we wrap up this section here.

If you are eager to learn more linear algebra, there are many excellent books and online resources. For a more advanced crash course, consider checking out Strang (1993), Kolter (2008), and Petersen and Pedersen (2008).

To recap:

- Scalars, vectors, matrices, and tensors are the basic mathematical objects used in linear algebra and have zero, one, two, and an arbitrary number of axes, respectively.
- Tensors can be sliced or reduced along specified axes via indexing, or operations such as `sum` and `mean`, respectively.

- Elementwise products are called Hadamard products. By contrast, dot products, matrix–vector products, and matrix–matrix products are not elementwise operations and in general return objects having shapes that are different from the the operands.
- Compared to Hadamard products, matrix–matrix products take considerably longer to compute (cubic rather than quadratic time).
- Norms capture various notions of the magnitude of a vector (or matrix), and are commonly applied to the difference of two vectors to measure their distance apart.
- Common vector norms include the ℓ_1 and ℓ_2 norms, and common matrix norms include the *spectral* and *Frobenius* norms.

2.3.13 Exercises

1. Prove that the transpose of the transpose of a matrix is the matrix itself: $(\mathbf{A}^\top)^\top = \mathbf{A}$.
2. Given two matrices $\mathbf{A}$ and $\mathbf{B}$, show that sum and transposition commute: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$.
3. Given any square matrix $\mathbf{A}$, is $\mathbf{A} + \mathbf{A}^\top$ always symmetric? Can you prove the result by using only the results of the previous two exercises?
4. We defined the tensor `X` of shape (2, 3, 4) in this section. What is the output of `len(X)`? Write your answer without implementing any code, then check your answer using code.
5. For a tensor `X` of arbitrary shape, does `len(X)` always correspond to the length of a certain axis of `X`? What is that axis?
6. Run `A / A.sum(axis=1)` and see what happens. Can you analyze the results?
7. When traveling between two points in downtown Manhattan, what is the distance that you need to cover in terms of the coordinates, i.e., in terms of avenues and streets? Can you travel diagonally?
8. Consider a tensor of shape (2, 3, 4). What are the shapes of the summation outputs along axes 0, 1, and 2?
9. Feed a tensor with three or more axes to the `linalg.norm` function and observe its output. What does this function compute for tensors of arbitrary shape?
10. Consider three large matrices, say $\mathbf{A} \in \mathbb{R}^{2^{10} \times 2^{16}}$, $\mathbf{B} \in \mathbb{R}^{2^{16} \times 2^5}$ and $\mathbf{C} \in \mathbb{R}^{2^5 \times 2^{14}}$, initialized with Gaussian random variables. You want to compute the product $\mathbf{ABC}$. Is there any difference in memory footprint and speed, depending on whether you compute $(\mathbf{AB})\mathbf{C}$ or $\mathbf{A}(\mathbf{BC})$. Why?
11. Consider three large matrices, say $\mathbf{A} \in \mathbb{R}^{2^{10} \times 2^{16}}$, $\mathbf{B} \in \mathbb{R}^{2^{16} \times 2^5}$ and $\mathbf{C} \in \mathbb{R}^{2^5 \times 2^{16}}$. Is there any difference in speed depending on whether you compute $\mathbf{AB}$ or $\mathbf{AC}^\top$? Why? What changes if you initialize $\mathbf{C} = \mathbf{B}^\top$ without cloning memory? Why?
12. Consider three matrices, say $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{100 \times 200}$. Construct a tensor with three axes by

stacking $[\mathbf{A}, \mathbf{B}, \mathbf{C}]$. What is the dimensionality? Slice out the second coordinate of the third axis to recover $\mathbf{B}$. Check that your answer is correct.

Discussions[55].

55 

2.4 Calculus

For a long time, how to calculate the area of a circle remained a mystery. Then, in Ancient Greece, the mathematician Archimedes came up with the clever idea to inscribe a series of polygons with increasing numbers of vertices on the inside of a circle (Fig. 2.4.1). For a polygon with n vertices, we obtain n triangles. The height of each triangle approaches the radius r as we partition the circle more finely. At the same time, its base approaches $2\pi r/n$, since the ratio between arc and secant approaches 1 for a large number of vertices. Thus, the area of the polygon approaches $n \cdot r \cdot \frac{1}{2}(2\pi r/n) = \pi r^2$.

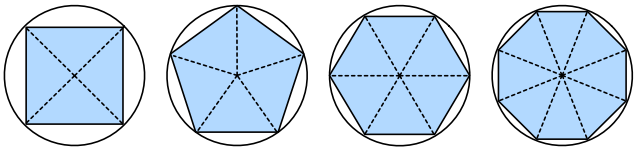


Fig. 2.4.1 Finding the area of a circle as a limit procedure.

This limiting procedure is at the root of both *differential calculus* and *integral calculus*. The former can tell us how to increase or decrease a function's value by manipulating its arguments. This comes in handy for the *optimization problems* that we face in deep learning, where we repeatedly update our parameters in order to decrease the loss function. Optimization addresses how to fit our models to training data, and calculus is its key prerequisite. However, do not forget that our ultimate goal is to perform well on *previously unseen* data. That problem is called *generalization* and will be a key focus of other chapters.

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

2.4.1 Derivatives and Differentiation

Put simply, a *derivative* is the rate of change in a function with respect to changes in its arguments. Derivatives can tell us how rapidly a loss function would increase or decrease were we to *increase* or *decrease* each parameter by an infinitesimally small amount. Formally, for functions $f : \mathbb{R} \to \mathbb{R}$, that map from scalars to scalars, the *derivative* of f at a point x is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{2.4.1}$$

This term on the right hand side is called a *limit* and it tells us what happens to the value of an expression as a specified variable approaches a particular value. This limit tells us what the ratio between a perturbation h and the change in the function value $f(x + h) - f(x)$ converges to as we shrink its size to zero.

When $f'(x)$ exists, f is said to be *differentiable* at x; and when $f'(x)$ exists for all x on a set, e.g., the interval $[a, b]$, we say that f is differentiable on this set. Not all functions are differentiable, including many that we wish to optimize, such as accuracy and the area under the receiving operating characteristic (AUC). However, because computing the derivative of the loss is a crucial step in nearly all algorithms for training deep neural networks, we often optimize a differentiable *surrogate* instead.

We can interpret the derivative $f'(x)$ as the *instantaneous* rate of change of $f(x)$ with respect to x. Let's develop some intuition with an example. Define $u = f(x) = 3x^2 - 4x$.

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

Setting $x = 1$, we see that $\frac{f(x+h)-f(x)}{h}$ approaches 2 as h approaches 0. While this experiment lacks the rigor of a mathematical proof, we can quickly see that indeed $f'(1) = 2$.

```
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
```

There are several equivalent notational conventions for derivatives. Given $y = f(x)$, the following expressions are equivalent:

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx}f(x) = Df(x) = D_x f(x), \tag{2.4.2}$$

where the symbols $\frac{d}{dx}$ and D are *differentiation operators*. Below, we present the derivatives of some common functions:

$$\begin{aligned}
&\frac{d}{dx}C = 0 && \text{for any constant } C \\
&\frac{d}{dx}x^n = nx^{n-1} && \text{for } n \neq 0 \\
&\frac{d}{dx}e^x = e^x \\
&\frac{d}{dx}\ln x = x^{-1}.
\end{aligned} \tag{2.4.3}$$

Functions composed from differentiable functions are often themselves differentiable. The following rules come in handy for working with compositions of any differentiable functions f and g, and constant C.

$$\begin{aligned}
\frac{d}{dx}[Cf(x)] &= C\frac{d}{dx}f(x) && \text{Constant multiple rule}\\
\frac{d}{dx}[f(x)+g(x)] &= \frac{d}{dx}f(x)+\frac{d}{dx}g(x) && \text{Sum rule}\\
\frac{d}{dx}[f(x)g(x)] &= f(x)\frac{d}{dx}g(x)+g(x)\frac{d}{dx}f(x) && \text{Product rule}\\
\frac{d}{dx}\frac{f(x)}{g(x)} &= \frac{g(x)\frac{d}{dx}f(x)-f(x)\frac{d}{dx}g(x)}{g^2(x)} && \text{Quotient rule}
\end{aligned} \tag{2.4.4}$$

Using this, we can apply the rules to find the derivative of $3x^2-4x$ via

$$\frac{d}{dx}[3x^2-4x] = 3\frac{d}{dx}x^2-4\frac{d}{dx}x = 6x-4. \tag{2.4.5}$$

Plugging in $x = 1$ shows that, indeed, the derivative equals 2 at this location. Note that derivatives tell us the *slope* of a function at a particular location.

2.4.2 Visualization Utilities

We can visualize the slopes of functions using the `matplotlib` library. We need to define a few functions. As its name indicates, `use_svg_display` tells `matplotlib` to output graphics in SVG format for crisper images. The comment `#@save` is a special modifier that allows us to save any function, class, or other code block to the `d2l` package so that we can invoke it later without repeating the code, e.g., via `d2l.use_svg_display()`.

```
def use_svg_display():  #@save
    """Use the svg format to display a plot in Jupyter."""
    backend_inline.set_matplotlib_formats('svg')
```

Conveniently, we can set figure sizes with `set_figsize`. Since the import statement `from matplotlib import pyplot as plt` was marked via `#@save` in the `d2l` package, we can call `d2l.plt`.

```
def set_figsize(figsize=(3.5, 2.5)):  #@save
    """Set the figure size for matplotlib."""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

The `set_axes` function can associate axes with properties, including labels, ranges, and scales.

```
#@save
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """Set the axes for matplotlib."""
    axes.set_xlabel(xlabel), axes.set_ylabel(ylabel)
```

(continues on next page)

(continued from previous page)

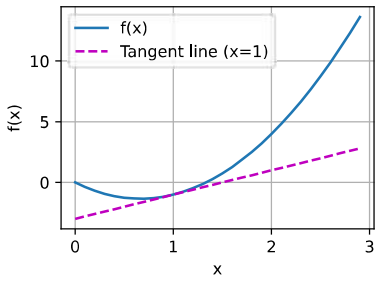
```
    axes.set_xscale(xscale), axes.set_yscale(yscale)
    axes.set_xlim(xlim),     axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()
```

With these three functions, we can define a `plot` function to overlay multiple curves. Much of the code here is just ensuring that the sizes and shapes of inputs match.

```
#@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):
    """Plot data points."""

    def has_one_axis(X):  # True if X (tensor or list) has 1 axis
        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)
```

Now we can plot the function $u = f(x)$ and its tangent line $y = 2x - 3$ at $x = 1$, where the coefficient 2 is the slope of the tangent line.

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



2.4.3 Partial Derivatives and Gradients

Thus far, we have been differentiating functions of just one variable. In deep learning, we also need to work with functions of *many* variables. We briefly introduce notions of the derivative that apply to such *multivariate* functions.

Let $y = f(x_1, x_2, \ldots, x_n)$ be a function with n variables. The *partial derivative* of y with respect to its i^{th} parameter x_i is

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{2.4.6}$$

To calculate $\frac{\partial y}{\partial x_i}$, we can treat $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ as constants and calculate the derivative of y with respect to x_i. The following notational conventions for partial derivatives are all common and all mean the same thing:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = \partial_{x_i} f = \partial_i f = f_{x_i} = f_i = D_i f = D_{x_i} f. \tag{2.4.7}$$

We can concatenate partial derivatives of a multivariate function with respect to all its variables to obtain a vector that is called the *gradient* of the function. Suppose that the input of function $f : \mathbb{R}^n \to \mathbb{R}$ is an n-dimensional vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$ and the output is a scalar. The gradient of the function f with respect to $\mathbf{x}$ is a vector of n partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\partial_{x_1} f(\mathbf{x}), \partial_{x_2} f(\mathbf{x}), \ldots \partial_{x_n} f(\mathbf{x})\right]^\top. \tag{2.4.8}$$

When there is no ambiguity, $\nabla_{\mathbf{x}} f(\mathbf{x})$ is typically replaced by $\nabla f(\mathbf{x})$. The following rules come in handy for differentiating multivariate functions:

- For all $\mathbf{A} \in \mathbb{R}^{m \times n}$ we have $\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$ and $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$.
- For square matrices $\mathbf{A} \in \mathbb{R}^{n \times n}$ we have that $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$ and in particular $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$.

Similarly, for any matrix $\mathbf{X}$, we have $\nabla_{\mathbf{X}} \|\mathbf{X}\|_{\text{F}}^2 = 2\mathbf{X}$.

2.4.4 Chain Rule

In deep learning, the gradients of concern are often difficult to calculate because we are working with deeply nested functions (of functions (of functions…)). Fortunately, the *chain rule* takes care of this. Returning to functions of a single variable, suppose that $y = f(g(x))$ and that the underlying functions $y = f(u)$ and $u = g(x)$ are both differentiable. The chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}. \tag{2.4.9}$$

Turning back to multivariate functions, suppose that $y = f(\mathbf{u})$ has variables $u_1, u_2, \ldots, u_m$, where each $u_i = g_i(\mathbf{x})$ has variables $x_1, x_2, \ldots, x_n$, i.e., $\mathbf{u} = g(\mathbf{x})$. Then the chain rule states that

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial u_1} \frac{\partial u_1}{\partial x_i} + \frac{\partial y}{\partial u_2} \frac{\partial u_2}{\partial x_i} + \ldots + \frac{\partial y}{\partial u_m} \frac{\partial u_m}{\partial x_i} \text{ and so } \nabla_{\mathbf{x}} y = \mathbf{A} \nabla_{\mathbf{u}} y, \tag{2.4.10}$$

where $\mathbf{A} \in \mathbb{R}^{n \times m}$ is a *matrix* that contains the derivative of vector $\mathbf{u}$ with respect to vector $\mathbf{x}$. Thus, evaluating the gradient requires computing a vector–matrix product. This is one of the key reasons why linear algebra is such an integral building block in building deep learning systems.

2.4.5 Discussion

While we have just scratched the surface of a deep topic, a number of concepts already come into focus: first, the composition rules for differentiation can be applied routinely, enabling us to compute gradients *automatically*. This task requires no creativity and thus we can focus our cognitive powers elsewhere. Second, computing the derivatives of vector-valued functions requires us to multiply matrices as we trace the dependency graph of variables from output to input. In particular, this graph is traversed in a *forward* direction when we evaluate a function and in a *backwards* direction when we compute gradients. Later chapters will formally introduce backpropagation, a computational procedure for applying the chain rule.

From the viewpoint of optimization, gradients allow us to determine how to move the parameters of a model in order to lower the loss, and each step of the optimization algorithms used throughout this book will require calculating the gradient.

2.4.6 Exercises

1. So far we took the rules for derivatives for granted. Using the definition and limits prove the properties for (i) $f(x) = c$, (ii) $f(x) = x^n$, (iii) $f(x) = e^x$ and (iv) $f(x) = \log x$.
2. In the same vein, prove the product, sum, and quotient rule from first principles.
3. Prove that the constant multiple rule follows as a special case of the product rule.
4. Calculate the derivative of $f(x) = x^x$.
5. What does it mean that $f'(x) = 0$ for some x? Give an example of a function f and a location x for which this might hold.
6. Plot the function $y = f(x) = x^3 - \frac{1}{x}$ and plot its tangent line at $x = 1$.
7. Find the gradient of the function $f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$.
8. What is the gradient of the function $f(\mathbf{x}) = \|\mathbf{x}\|_2$? What happens for $\mathbf{x} = \mathbf{0}$?
9. Can you write out the chain rule for the case where $u = f(x, y, z)$ and $x = x(a, b)$, $y = y(a, b)$, and $z = z(a, b)$?
10. Given a function $f(x)$ that is invertible, compute the derivative of its inverse $f^{-1}(x)$. Here we have that $f^{-1}(f(x)) = x$ and conversely $f(f^{-1}(y)) = y$. Hint: use these properties in your derivation.

Discussions[56].

2.5 Automatic Differentiation

Recall from Section 2.4 that calculating derivatives is the crucial step in all the optimization algorithms that we will use to train deep networks. While the calculations are straightforward, working them out by hand can be tedious and error-prone, and these issues only grow as our models become more complex.

Fortunately all modern deep learning frameworks take this work off our plates by offering *automatic differentiation* (often shortened to *autograd*). As we pass data through each successive function, the framework builds a *computational graph* that tracks how each value depends on others. To calculate derivatives, automatic differentiation works backwards through this graph applying the chain rule. The computational algorithm for applying the chain rule in this fashion is called *backpropagation*.

While autograd libraries have become a hot concern over the past decade, they have a long history. In fact the earliest references to autograd date back over half of a century (Wengert, 1964). The core ideas behind modern backpropagation date to a PhD thesis from 1980 (Speelpenning, 1980) and were further developed in the late 1980s (Griewank, 1989). While backpropagation has become the default method for computing gradients, it is not the only option. For instance, the Julia programming language employs forward propagation (Revels *et al.*, 2016). Before exploring methods, let's first master the autograd package.

```
import torch
```

2.5.1 A Simple Function

Let's assume that we are interested in differentiating the function $y = 2\mathbf{x}^\top \mathbf{x}$ with respect to the column vector $\mathbf{x}$. To start, we assign `x` an initial value.

```
x = torch.arange(4.0)
x
```

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

Before we calculate the gradient of y with respect to $\mathbf{x}$, we need a place to store it. In general, we avoid allocating new memory every time we take a derivative because deep learning requires successively computing derivatives with respect to the same parameters a great many times, and we might risk running out of memory. Note that the gradient of a scalar-valued function with respect to a vector $\mathbf{x}$ is vector-valued with the same shape as $\mathbf{x}$.

```
# Can also create x = torch.arange(4.0, requires_grad=True)
x.requires_grad_(True)
x.grad  # The gradient is None by default
```

We now calculate our function of x and assign the result to y.

```
y = 2 * torch.dot(x, x)
y
```

```
tensor(28., grad_fn=<MulBackward0>)
```

We can now take the gradient of y with respect to x by calling its backward method. Next, we can access the gradient via x's grad attribute.

```
y.backward()
x.grad
```

```
tensor([ 0.,  4.,  8., 12.])
```

We already know that the gradient of the function $y = 2\mathbf{x}^\top\mathbf{x}$ with respect to $\mathbf{x}$ should be $4\mathbf{x}$. We can now verify that the automatic gradient computation and the expected result are identical.

```
x.grad == 4 * x
```

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

Now let's calculate another function of x and take its gradient. Note that PyTorch does not automatically reset the gradient buffer when we record a new gradient. Instead, the new gradient is added to the already-stored gradient. This behavior comes in handy when we want to optimize the sum of multiple objective functions. To reset the gradient buffer, we can call x.grad.zero_() as follows:

```
x.grad.zero_()  # Reset the gradient
y = x.sum()
y.backward()
x.grad
```

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

2.5.2 Backward for Non-Scalar Variables

When y is a vector, the most natural representation of the derivative of y with respect to a vector x is a matrix called the *Jacobian* that contains the partial derivatives of each

component of y with respect to each component of x. Likewise, for higher-order y and x, the result of differentiation could be an even higher-order tensor.

While Jacobians do show up in some advanced machine learning techniques, more commonly we want to sum up the gradients of each component of y with respect to the full vector x, yielding a vector of the same shape as x. For example, we often have a vector representing the value of our loss function calculated separately for each example among a *batch* of training examples. Here, we just want to sum up the gradients computed individually for each example.

Because deep learning frameworks vary in how they interpret gradients of non-scalar tensors, PyTorch takes some steps to avoid confusion. Invoking `backward` on a non-scalar elicits an error unless we tell PyTorch how to reduce the object to a scalar. More formally, we need to provide some vector $\mathbf{v}$ such that `backward` will compute $\mathbf{v}^\top \partial_{\mathbf{x}} \mathbf{y}$ rather than $\partial_{\mathbf{x}} \mathbf{y}$. This next part may be confusing, but for reasons that will become clear later, this argument (representing $\mathbf{v}$) is named `gradient`. For a more detailed description, see Yang Zhang's Medium post[57].

57 

```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))  # Faster: y.sum().backward()
x.grad
```

```
tensor([0., 2., 4., 6.])
```

2.5.3 Detaching Computation

Sometimes, we wish to move some calculations outside of the recorded computational graph. For example, say that we use the input to create some auxiliary intermediate terms for which we do not want to compute a gradient. In this case, we need to *detach* the respective computational graph from the final result. The following toy example makes this clearer: suppose we have `z = x * y` and `y = x * x` but we want to focus on the *direct* influence of x on z rather than the influence conveyed via y. In this case, we can create a new variable u that takes the same value as y but whose *provenance* (how it was created) has been wiped out. Thus u has no ancestors in the graph and gradients do not flow through u to x. For example, taking the gradient of `z = x * u` will yield the result u, (not `3 * x * x * x` as you might have expected since `z = x * x * x * x`).

```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

z.sum().backward()
x.grad == u
```

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

Note that while this procedure detaches `y`'s ancestors from the graph leading to `z`, the computational graph leading to `y` persists and thus we can calculate the gradient of `y` with respect to `x`.

```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

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

2.5.4 Gradients and Python Control Flow

So far we reviewed cases where the path from input to output was well defined via a function such as `z = x * x * x`. Programming offers us a lot more freedom in how we compute results. For instance, we can make them depend on auxiliary variables or condition choices on intermediate results. One benefit of using automatic differentiation is that even if building the computational graph of a function required passing through a maze of Python control flow (e.g., conditionals, loops, and arbitrary function calls), we can still calculate the gradient of the resulting variable. To illustrate this, consider the following code snippet where the number of iterations of the `while` loop and the evaluation of the `if` statement both depend on the value of the input `a`.

```
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
```

Below, we call this function, passing in a random value, as input. Since the input is a random variable, we do not know what form the computational graph will take. However, whenever we execute `f(a)` on a specific input, we realize a specific computational graph and can subsequently run `backward`.

```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

Even though our function `f` is, for demonstration purposes, a bit contrived, its dependence on the input is quite simple: it is a *linear* function of `a` with piecewise defined scale. As

such, `f(a) / a` is a vector of constant entries and, moreover, `f(a) / a` needs to match the gradient of `f(a)` with respect to `a`.

```
a.grad == d / a
```

```
tensor(True)
```

Dynamic control flow is very common in deep learning. For instance, when processing text, the computational graph depends on the length of the input. In these cases, automatic differentiation becomes vital for statistical modeling since it is impossible to compute the gradient *a priori*.

2.5.5 Discussion

You have now gotten a taste of the power of automatic differentiation. The development of libraries for calculating derivatives both automatically and efficiently has been a massive productivity booster for deep learning practitioners, liberating them so they can focus on less menial. Moreover, autograd lets us design massive models for which pen and paper gradient computations would be prohibitively time consuming. Interestingly, while we use autograd to *optimize* models (in a statistical sense) the *optimization* of autograd libraries themselves (in a computational sense) is a rich subject of vital interest to framework designers. Here, tools from compilers and graph manipulation are leveraged to compute results in the most expedient and memory-efficient manner.

For now, try to remember these basics: (i) attach gradients to those variables with respect to which we desire derivatives; (ii) record the computation of the target value; (iii) execute the backpropagation function; and (iv) access the resulting gradient.

2.5.6 Exercises

1. Why is the second derivative much more expensive to compute than the first derivative?
2. After running the function for backpropagation, immediately run it again and see what happens. Investigate.
3. In the control flow example where we calculate the derivative of `d` with respect to `a`, what would happen if we changed the variable `a` to a random vector or a matrix? At this point, the result of the calculation `f(a)` is no longer a scalar. What happens to the result? How do we analyze this?
4. Let $f(x) = \sin(x)$. Plot the graph of f and of its derivative f'. Do not exploit the fact that $f'(x) = \cos(x)$ but rather use automatic differentiation to get the result.
5. Let $f(x) = ((\log x^2) \cdot \sin x) + x^{-1}$. Write out a dependency graph tracing results from x to $f(x)$.
6. Use the chain rule to compute the derivative $\frac{df}{dx}$ of the aforementioned function, placing each term on the dependency graph that you constructed previously.

7. Given the graph and the intermediate derivative results, you have a number of options when computing the gradient. Evaluate the result once starting from x to f and once from f tracing back to x. The path from x to f is commonly known as *forward differentiation*, whereas the path from f to x is known as backward differentiation.

8. When might you want to use forward, and when backward, differentiation? Hint: consider the amount of intermediate data needed, the ability to parallelize steps, and the size of matrices and vectors involved.

Discussions[58].

[58]

2.6 Probability and Statistics

One way or another, machine learning is all about uncertainty. In supervised learning, we want to predict something unknown (the *target*) given something known (the *features*). Depending on our objective, we might attempt to predict the most likely value of the target. Or we might predict the value with the smallest expected distance from the target. And sometimes we wish not only to predict a specific value but to *quantify our uncertainty*. For example, given some features describing a patient, we might want to know *how likely* they are to suffer a heart attack in the next year. In unsupervised learning, we often care about uncertainty. To determine whether a set of measurements are anomalous, it helps to know how likely one is to observe values in a population of interest. Furthermore, in reinforcement learning, we wish to develop agents that act intelligently in various environments. This requires reasoning about how an environment might be expected to change and what rewards one might expect to encounter in response to each of the available actions.

Probability is the mathematical field concerned with reasoning under uncertainty. Given a probabilistic model of some process, we can reason about the likelihood of various events. The use of probabilities to describe the frequencies of repeatable events (like coin tosses) is fairly uncontroversial. In fact, *frequentist* scholars adhere to an interpretation of probability that applies *only* to such repeatable events. By contrast *Bayesian* scholars use the language of probability more broadly to formalize reasoning under uncertainty. Bayesian probability is characterized by two unique features: (i) assigning degrees of belief to non-repeatable events, e.g., what is the *probability* that a dam will collapse?; and (ii) subjectivity. While Bayesian probability provides unambiguous rules for how one should update their beliefs in light of new evidence, it allows for different individuals to start off with different *prior* beliefs. *Statistics* helps us to reason backwards, starting off with collection and organization of data and backing out to what inferences we might draw about the process that generated the data. Whenever we analyze a dataset, hunting for patterns that we hope might characterize a broader population, we are employing statistical thinking. Many courses, majors, theses, careers, departments, companies, and institutions have been devoted to the study of probability and statistics. While this section only scratches the surface, we will provide the foundation that you need to begin building models.

```
%matplotlib inline
import random
import torch
from torch.distributions.multinomial import Multinomial
from d2l import torch as d2l
```

2.6.1 A Simple Example: Tossing Coins

Imagine that we plan to toss a coin and want to quantify how likely we are to see heads (vs. tails). If the coin is *fair*, then both outcomes (heads and tails), are equally likely. Moreover if we plan to toss the coin n times then the fraction of heads that we *expect* to see should exactly match the *expected* fraction of tails. One intuitive way to see this is by symmetry: for every possible outcome with n_h heads and $n_\mathrm{t} = (n - n_\mathrm{h})$ tails, there is an equally likely outcome with n_t heads and n_h tails. Note that this is only possible if on average we expect to see $1/2$ of tosses come up heads and $1/2$ come up tails. Of course, if you conduct this experiment many times with $n = 1000000$ tosses each, you might never see a trial where $n_\mathrm{h} = n_\mathrm{t}$ exactly.

Formally, the quantity $1/2$ is called a *probability* and here it captures the certainty with which any given toss will come up heads. Probabilities assign scores between 0 and 1 to outcomes of interest, called *events*. Here the event of interest is heads and we denote the corresponding probability $P(\text{heads})$. A probability of 1 indicates absolute certainty (imagine a trick coin where both sides were heads) and a probability of 0 indicates impossibility (e.g., if both sides were tails). The frequencies n_h/n and n_t/n are not probabilities but rather *statistics*. Probabilities are *theoretical* quantities that underly the data generating process. Here, the probability $1/2$ is a property of the coin itself. By contrast, statistics are *empirical* quantities that are computed as functions of the observed data. Our interests in probabilistic and statistical quantities are inextricably intertwined. We often design special statistics called *estimators* that, given a dataset, produce *estimates* of model parameters such as probabilities. Moreover, when those estimators satisfy a nice property called *consistency*, our estimates will converge to the corresponding probability. In turn, these inferred probabilities tell about the likely statistical properties of data from the same population that we might encounter in the future.

Suppose that we stumbled upon a real coin for which we did not know the true $P(\text{heads})$. To investigate this quantity with statistical methods, we need to (i) collect some data; and (ii) design an estimator. Data acquisition here is easy; we can toss the coin many times and record all the outcomes. Formally, drawing realizations from some underlying random process is called *sampling*. As you might have guessed, one natural estimator is the ratio of the number of observed *heads* to the total number of tosses.

Now, suppose that the coin was in fact fair, i.e., $P(\text{heads}) = 0.5$. To simulate tosses of a fair coin, we can invoke any random number generator. There are some easy ways to draw samples of an event with probability 0.5. For example Python's `random.random` yields numbers in the interval $[0, 1]$ where the probability of lying in any sub-interval $[a, b] \subset$

$[0, 1]$ is equal to $b - a$. Thus we can get out `0` and `1` with probability `0.5` each by testing whether the returned float number is greater than `0.5`:

```
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:  [44, 56]
```

More generally, we can simulate multiple draws from any variable with a finite number of possible outcomes (like the toss of a coin or roll of a die) by calling the multinomial function, setting the first argument to the number of draws and the second as a list of probabilities associated with each of the possible outcomes. To simulate ten tosses of a fair coin, we assign probability vector `[0.5, 0.5]`, interpreting index 0 as heads and index 1 as tails. The function returns a vector with length equal to the number of possible outcomes (here, 2), where the first component tells us the number of occurrences of heads and the second component tells us the number of occurrences of tails.

```
fair_probs = torch.tensor([0.5, 0.5])
Multinomial(100, fair_probs).sample()
```

```
tensor([50., 50.])
```

Each time you run this sampling process, you will receive a new random value that may differ from the previous outcome. Dividing by the number of tosses gives us the *frequency* of each outcome in our data. Note that these frequencies, just like the probabilities that they are intended to estimate, sum to 1.

```
Multinomial(100, fair_probs).sample() / 100
```

```
tensor([0.4800, 0.5200])
```

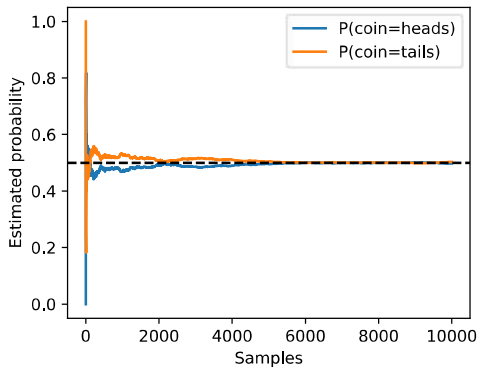
Here, even though our simulated coin is fair (we ourselves set the probabilities `[0.5, 0.5]`), the counts of heads and tails may not be identical. That is because we only drew a relatively small number of samples. If we did not implement the simulation ourselves, and only saw the outcome, how would we know if the coin were slightly unfair or if the possible deviation from $1/2$ was just an artifact of the small sample size? Let's see what happens when we simulate 10,000 tosses.

```
counts = Multinomial(10000, fair_probs).sample()
counts / 10000
```

```
tensor([0.4966, 0.5034])
```

In general, for averages of repeated events (like coin tosses), as the number of repetitions grows, our estimates are guaranteed to converge to the true underlying probabilities. The mathematical formulation of this phenomenon is called the *law of large numbers* and the *central limit theorem* tells us that in many situations, as the sample size n grows, these errors should go down at a rate of $(1/\sqrt{n})$. Let's get some more intuition by studying how our estimate evolves as we grow the number of tosses from 1 to 10,000.

```
counts = Multinomial(1, fair_probs).sample((10000,))
cum_counts = counts.cumsum(dim=0)
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();
```



Each solid curve corresponds to one of the two values of the coin and gives our estimated probability that the coin turns up that value after each group of experiments. The dashed black line gives the true underlying probability. As we get more data by conducting more experiments, the curves converge towards the true probability. You might already begin to see the shape of some of the more advanced questions that preoccupy statisticians: How quickly does this convergence happen? If we had already tested many coins manufactured at the same plant, how might we incorporate this information?

2.6.2 A More Formal Treatment

We have already gotten pretty far: posing a probabilistic model, generating synthetic data, running a statistical estimator, empirically assessing convergence, and reporting error met-

rics (checking the deviation). However, to go much further, we will need to be more precise.

When dealing with randomness, we denote the set of possible outcomes $\mathcal{S}$ and call it the *sample space* or *outcome space*. Here, each element is a distinct possible *outcome*. In the case of rolling a single coin, $\mathcal{S} = \{\text{heads}, \text{tails}\}$. For a single die, $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$. When flipping two coins, possible outcomes are $\{(\text{heads}, \text{heads}), (\text{heads}, \text{tails}), (\text{tails}, \text{heads}), (\text{tails}, \text{tails})\}$. *Events* are subsets of the sample space. For instance, the event "the first coin toss comes up heads" corresponds to the set $\{(\text{heads}, \text{heads}), (\text{heads}, \text{tails})\}$. Whenever the outcome z of a random experiment satisfies $z \in \mathcal{A}$, then event $\mathcal{A}$ has occurred. For a single roll of a die, we could define the events "seeing a 5" ($\mathcal{A} = \{5\}$) and "seeing an odd number" ($\mathcal{B} = \{1, 3, 5\}$). In this case, if the die came up 5, we would say that both $\mathcal{A}$ and $\mathcal{B}$ occurred. On the other hand, if $z = 3$, then $\mathcal{A}$ did not occur but $\mathcal{B}$ did.

A *probability* function maps events onto real values $P : \mathcal{A} \subseteq \mathcal{S} \rightarrow [0, 1]$. The probability, denoted $P(\mathcal{A})$, of an event $\mathcal{A}$ in the given sample space $\mathcal{S}$, has the following properties:

- The probability of any event $\mathcal{A}$ is a nonnegative real number, i.e., $P(\mathcal{A}) \geq 0$;
- The probability of the entire sample space is 1, i.e., $P(\mathcal{S}) = 1$;
- For any countable sequence of events $\mathcal{A}_1, \mathcal{A}_2, \ldots$ that are *mutually exclusive* (i.e., $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ for all $i \neq j$), the probability that any of them happens is equal to the sum of their individual probabilities, i.e., $P(\bigcup_{i=1}^{\infty} \mathcal{A}_i) = \sum_{i=1}^{\infty} P(\mathcal{A}_i)$.

These axioms of probability theory, proposed by Kolmogorov (1933), can be applied to rapidly derive a number of important consequences. For instance, it follows immediately that the probability of any event $\mathcal{A}$ *or* its complement $\mathcal{A}'$ occurring is 1 (because $\mathcal{A} \cup \mathcal{A}' = \mathcal{S}$). We can also prove that $P(\emptyset) = 0$ because $1 = P(\mathcal{S} \cup \mathcal{S}') = P(\mathcal{S} \cup \emptyset) = P(\mathcal{S}) + P(\emptyset) = 1 + P(\emptyset)$. Consequently, the probability of any event $\mathcal{A}$ *and* its complement $\mathcal{A}'$ occurring simultaneously is $P(\mathcal{A} \cap \mathcal{A}') = 0$. Informally, this tells us that impossible events have zero probability of occurring.

2.6.3 Random Variables

When we spoke about events like the roll of a die coming up odds or the first coin toss coming up heads, we were invoking the idea of a *random variable*. Formally, random variables are mappings from an underlying sample space to a set of (possibly many) values. You might wonder how a random variable is different from the sample space, since both are collections of outcomes. Importantly, random variables can be much coarser than the raw sample space. We can define a binary random variable like "greater than 0.5" even when the underlying sample space is infinite, e.g., points on the line segment between 0 and 1. Additionally, multiple random variables can share the same underlying sample space. For example "whether my home alarm goes off" and "whether my house was burgled" are both binary random variables that share an underlying sample space. Consequently, knowing the value taken by one random variable can tell us something about the likely value of another

random variable. Knowing that the alarm went off, we might suspect that the house was likely burgled.

Every value taken by a random variable corresponds to a subset of the underlying sample space. Thus the occurrence where the random variable X takes value v, denoted by $X = v$, is an *event* and $P(X = v)$ denotes its probability. Sometimes this notation can get clunky, and we can abuse notation when the context is clear. For example, we might use $P(X)$ to refer broadly to the *distribution* of X, i.e., the function that tells us the probability that X takes any given value. Other times we write expressions like $P(X, Y) = P(X)P(Y)$, as a shorthand to express a statement that is true for all of the values that the random variables X and Y can take, i.e., for all i, j it holds that $P(X = i \text{ and } Y = j) = P(X = i)P(Y = j)$. Other times, we abuse notation by writing $P(v)$ when the random variable is clear from the context. Since an event in probability theory is a set of outcomes from the sample space, we can specify a range of values for a random variable to take. For example, $P(1 \leq X \leq 3)$ denotes the probability of the event $\{1 \leq X \leq 3\}$.

Note that there is a subtle difference between *discrete* random variables, like flips of a coin or tosses of a die, and *continuous* ones, like the weight and the height of a person sampled at random from the population. In this case we seldom really care about someone's exact height. Moreover, if we took precise enough measurements, we would find that no two people on the planet have the exact same height. In fact, with fine enough measurements, you would never have the same height when you wake up and when you go to sleep. There is little point in asking about the exact probability that someone is 1.801392782910287192 meters tall. Instead, we typically care more about being able to say whether someone's height falls into a given interval, say between 1.79 and 1.81 meters. In these cases we work with probability *densities*. The height of exactly 1.80 meters has no probability, but nonzero density. To work out the probability assigned to an interval, we must take an *integral* of the density over that interval.

2.6.4 Multiple Random Variables

You might have noticed that we could not even make it through the previous section without making statements involving interactions among multiple random variables (recall $P(X, Y) = P(X)P(Y)$). Most of machine learning is concerned with such relationships. Here, the sample space would be the population of interest, say customers who transact with a business, photographs on the Internet, or proteins known to biologists. Each random variable would represent the (unknown) value of a different attribute. Whenever we sample an individual from the population, we observe a realization of each of the random variables. Because the values taken by random variables correspond to subsets of the sample space that could be overlapping, partially overlapping, or entirely disjoint, knowing the value taken by one random variable can cause us to update our beliefs about which values of another random variable are likely. If a patient walks into a hospital and we observe that they are having trouble breathing and have lost their sense of smell, then we believe that they are more likely to have COVID-19 than we might if they had no trouble breathing and a perfectly ordinary sense of smell.

When working with multiple random variables, we can construct events corresponding to

every combination of values that the variables can jointly take. The probability function that assigns probabilities to each of these combinations (e.g. $A = a$ and $B = b$) is called the *joint probability* function and simply returns the probability assigned to the intersection of the corresponding subsets of the sample space. The *joint probability* assigned to the event where random variables A and B take values a and b, respectively, is denoted $P(A = a, B = b)$, where the comma indicates "and". Note that for any values a and b, it follows that

$$P(A = a, B = b) \leq P(A = a) \text{ and } P(A = a, B = b) \leq P(B = b), \tag{2.6.1}$$

since for $A = a$ and $B = b$ to happen, $A = a$ has to happen *and* $B = b$ also has to happen. Interestingly, the joint probability tells us all that we can know about these random variables in a probabilistic sense, and can be used to derive many other useful quantities, including recovering the individual distributions $P(A)$ and $P(B)$. To recover $P(A = a)$ we simply sum up $P(A = a, B = v)$ over all values v that the random variable B can take: $P(A = a) = \sum_v P(A = a, B = v)$.

The ratio $\frac{P(A=a,B=b)}{P(A=a)} \leq 1$ turns out to be extremely important. It is called the *conditional probability*, and is denoted via the "|" symbol:

$$P(B = b \mid A = a) = P(A = a, B = b)/P(A = a). \tag{2.6.2}$$

It tells us the new probability associated with the event $B = b$, once we condition on the fact $A = a$ took place. We can think of this conditional probability as restricting attention only to the subset of the sample space associated with $A = a$ and then renormalizing so that all probabilities sum to 1. Conditional probabilities are in fact just ordinary probabilities and thus respect all of the axioms, as long as we condition all terms on the same event and thus restrict attention to the same sample space. For instance, for disjoint events $\mathcal{B}$ and $\mathcal{B}'$, we have that $P(\mathcal{B} \cup \mathcal{B}' \mid A = a) = P(\mathcal{B} \mid A = a) + P(\mathcal{B}' \mid A = a)$.

Using the definition of conditional probabilities, we can derive the famous result called *Bayes' theorem*. By construction, we have that $P(A, B) = P(B \mid A)P(A)$ and $P(A, B) = P(A \mid B)P(B)$. Combining both equations yields $P(B \mid A)P(A) = P(A \mid B)P(B)$ and hence

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}. \tag{2.6.3}$$

This simple equation has profound implications because it allows us to reverse the order of conditioning. If we know how to estimate $P(B \mid A)$, $P(A)$, and $P(B)$, then we can estimate $P(A \mid B)$. We often find it easier to estimate one term directly but not the other and Bayes' theorem can come to the rescue here. For instance, if we know the prevalence of symptoms for a given disease, and the overall prevalences of the disease and symptoms, respectively, we can determine how likely someone is to have the disease based on their symptoms. In some cases we might not have direct access to $P(B)$, such as the prevalence of symptoms. In this case a simplified version of Bayes' theorem comes in handy:

$$P(A \mid B) \propto P(B \mid A)P(A). \tag{2.6.4}$$

Since we know that $P(A \mid B)$ must be normalized to 1, i.e., $\sum_a P(A = a \mid B) = 1$, we can use it to compute

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{\sum_a P(B \mid A = a)P(A = a)}. \tag{2.6.5}$$

In Bayesian statistics, we think of an observer as possessing some (subjective) prior beliefs about the plausibility of the available hypotheses encoded in the *prior* $P(H)$, and a *likelihood function* that says how likely one is to observe any value of the collected evidence for each of the hypotheses in the class $P(E \mid H)$. Bayes' theorem is then interpreted as telling us how to update the initial *prior* $P(H)$ in light of the available evidence E to produce *posterior* beliefs $P(H \mid E) = \frac{P(E|H)P(H)}{P(E)}$. Informally, this can be stated as "posterior equals prior times likelihood, divided by the evidence". Now, because the evidence $P(E)$ is the same for all hypotheses, we can get away with simply normalizing over the hypotheses.

Note that $\sum_a P(A = a \mid B) = 1$ also allows us to *marginalize* over random variables. That is, we can drop variables from a joint distribution such as $P(A, B)$. After all, we have that

$$\sum_a P(B \mid A = a)P(A = a) = \sum_a P(B, A = a) = P(B). \tag{2.6.6}$$

Independence is another fundamentally important concept that forms the backbone of many important ideas in statistics. In short, two variables are *independent* if conditioning on the value of A does not cause any change to the probability distribution associated with B and vice versa. More formally, independence, denoted $A \perp B$, requires that $P(A \mid B) = P(A)$ and, consequently, that $P(A, B) = P(A \mid B)P(B) = P(A)P(B)$. Independence is often an appropriate assumption. For example, if the random variable A represents the outcome from tossing one fair coin and the random variable B represents the outcome from tossing another, then knowing whether A came up heads should not influence the probability of B coming up heads.

Independence is especially useful when it holds among the successive draws of our data from some underlying distribution (allowing us to make strong statistical conclusions) or when it holds among various variables in our data, allowing us to work with simpler models that encode this independence structure. On the other hand, estimating the dependencies among random variables is often the very aim of learning. We care to estimate the probability of disease given symptoms specifically because we believe that diseases and symptoms are *not* independent.

Note that because conditional probabilities are proper probabilities, the concepts of independence and dependence also apply to them. Two random variables A and B are *conditionally independent* given a third variable C if and only if $P(A, B \mid C) = P(A \mid C)P(B \mid C)$. Interestingly, two variables can be independent in general but become dependent when conditioning on a third. This often occurs when the two random variables A and B correspond to causes of some third variable C. For example, broken bones and lung cancer might be independent in the general population but if we condition on being in the hospital then we might find that broken bones are negatively correlated with lung cancer. That is

because the broken bone *explains away* why some person is in the hospital and thus lowers the probability that they are hospitalized because of having lung cancer.

And conversely, two dependent random variables can become independent upon conditioning on a third. This often happens when two otherwise unrelated events have a common cause. Shoe size and reading level are highly correlated among elementary school students, but this correlation disappears if we condition on age.

2.6.5 An Example

Let's put our skills to the test. Assume that a doctor administers an HIV test to a patient. This test is fairly accurate and fails only with 1% probability if the patient is healthy but reported as diseased, i.e., healthy patients test positive in 1% of cases. Moreover, it never fails to detect HIV if the patient actually has it. We use $D_1 \in \{0, 1\}$ to indicate the diagnosis (0 if negative and 1 if positive) and $H \in \{0, 1\}$ to denote the HIV status.

Conditional probability	$H = 1$	$H = 0$
$P(D_1 = 1 \mid H)$	1	0.01
$P(D_1 = 0 \mid H)$	0	0.99

Note that the column sums are all 1 (but the row sums do not), since they are conditional probabilities. Let's compute the probability of the patient having HIV if the test comes back positive, i.e., $P(H = 1 \mid D_1 = 1)$. Intuitively this is going to depend on how common the disease is, since it affects the number of false alarms. Assume that the population is fairly free of the disease, e.g., $P(H = 1) = 0.0015$. To apply Bayes' theorem, we need to apply marginalization to determine

$$\begin{aligned} P(D_1 = 1) =& P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1) \\ =& P(D_1 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1 \mid H = 1)P(H = 1) \\ =& 0.011485. \end{aligned} \tag{2.6.7}$$

This leads us to

$$P(H = 1 \mid D_1 = 1) = \frac{P(D_1 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1)} = 0.1306. \tag{2.6.8}$$

In other words, there is only a 13.06% chance that the patient actually has HIV, despite the test being pretty accurate. As we can see, probability can be counterintuitive. What should a patient do upon receiving such terrifying news? Likely, the patient would ask the physician to administer another test to get clarity. The second test has different characteristics and it is not as good as the first one.

Conditional probability	$H = 1$	$H = 0$
$P(D_2 = 1 \mid H)$	0.98	0.03
$P(D_2 = 0 \mid H)$	0.02	0.97

Unfortunately, the second test comes back positive, too. Let's calculate the requisite probabilities to invoke Bayes' theorem by assuming conditional independence:

$$\begin{aligned} P(D_1 = 1, D_2 = 1 \mid H = 0) &= P(D_1 = 1 \mid H = 0)P(D_2 = 1 \mid H = 0) = 0.0003, \\ P(D_1 = 1, D_2 = 1 \mid H = 1) &= P(D_1 = 1 \mid H = 1)P(D_2 = 1 \mid H = 1) = 0.98. \end{aligned} \tag{2.6.9}$$

Now we can apply marginalization to obtain the probability that both tests come back positive:

$$\begin{aligned} &P(D_1 = 1, D_2 = 1) \\ &= P(D_1 = 1, D_2 = 1, H = 0) + P(D_1 = 1, D_2 = 1, H = 1) \\ &= P(D_1 = 1, D_2 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1) \\ &= 0.00176955. \end{aligned} \tag{2.6.10}$$

Finally, the probability of the patient having HIV given that both tests are positive is

$$P(H = 1 \mid D_1 = 1, D_2 = 1) = \frac{P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1, D_2 = 1)} = 0.8307. \tag{2.6.11}$$

That is, the second test allowed us to gain much higher confidence that not all is well. Despite the second test being considerably less accurate than the first one, it still significantly improved our estimate. The assumption of both tests being conditionally independent of each other was crucial for our ability to generate a more accurate estimate. Take the extreme case where we run the same test twice. In this situation we would expect the same outcome both times, hence no additional insight is gained from running the same test again. The astute reader might have noticed that the diagnosis behaved like a classifier hiding in plain sight where our ability to decide whether a patient is healthy increases as we obtain more features (test outcomes).

2.6.6 Expectations

Often, making decisions requires not just looking at the probabilities assigned to individual events but composing them together into useful aggregates that can provide us with guidance. For example, when random variables take continuous scalar values, we often care about knowing what value to expect *on average*. This quantity is formally called an *expectation*. If we are making investments, the first quantity of interest might be the return we can expect, averaging over all the possible outcomes (and weighting by the appropriate probabilities). For instance, say that with 50% probability, an investment might fail altogether, with 40% probability it might provide a 2× return, and with 10% probability it might provide a 10× return 10×. To calculate the expected return, we sum over all returns, multiplying each by the probability that they will occur. This yields the expectation $0.5 \cdot 0 + 0.4 \cdot 2 + 0.1 \cdot 10 = 1.8$. Hence the expected return is 1.8×.

In general, the *expectation* (or average) of the random variable X is defined as

$$E[X] = E_{x \sim P}[x] = \sum_x x P(X = x). \tag{2.6.12}$$

Likewise, for densities we obtain $E[X] = \int x \, dp(x)$. Sometimes we are interested in the expected value of some function of x. We can calculate these expectations as

$$E_{x\sim P}[f(x)] = \sum_x f(x)P(x) \text{ and } E_{x\sim P}[f(x)] = \int f(x)p(x) \, dx \tag{2.6.13}$$

for discrete probabilities and densities, respectively. Returning to the investment example from above, f might be the *utility* (happiness) associated with the return. Behavior economists have long noted that people associate greater disutility with losing money than the utility gained from earning one dollar relative to their baseline. Moreover, the value of money tends to be sub-linear. Possessing 100k dollars versus zero dollars can make the difference between paying the rent, eating well, and enjoying quality healthcare versus suffering through homelessness. On the other hand, the gains due to possessing 200k versus 100k are less dramatic. Reasoning like this motivates the cliché that "the utility of money is logarithmic".

If the utility associated with a total loss were -1, and the utilities associated with returns of 1, 2, and 10 were 1, 2 and 4, respectively, then the expected happiness of investing would be $0.5 \cdot (-1) + 0.4 \cdot 2 + 0.1 \cdot 4 = 0.7$ (an expected loss of utility of 30%). If indeed this were your utility function, you might be best off keeping the money in the bank.

For financial decisions, we might also want to measure how *risky* an investment is. Here, we care not just about the expected value but how much the actual values tend to *vary* relative to this value. Note that we cannot just take the expectation of the difference between the actual and expected values. This is because the expectation of a difference is the difference of the expectations, i.e., $E[X - E[X]] = E[X] - E[E[X]] = 0$. However, we can look at the expectation of any non-negative function of this difference. The *variance* of a random variable is calculated by looking at the expected value of the *squared* differences:

$$\text{Var}[X] = E\left[(X - E[X])^2\right] = E[X^2] - E[X]^2. \tag{2.6.14}$$

Here the equality follows by expanding $(X - E[X])^2 = X^2 - 2XE[X] + E[X]^2$ and taking expectations for each term. The square root of the variance is another useful quantity called the *standard deviation*. While this and the variance convey the same information (either can be calculated from the other), the standard deviation has the nice property that it is expressed in the same units as the original quantity represented by the random variable.

Lastly, the variance of a function of a random variable is defined analogously as

$$\text{Var}_{x\sim P}[f(x)] = E_{x\sim P}[f^2(x)] - E_{x\sim P}[f(x)]^2. \tag{2.6.15}$$

Returning to our investment example, we can now compute the variance of the investment. It is given by $0.5 \cdot 0 + 0.4 \cdot 2^2 + 0.1 \cdot 10^2 - 1.8^2 = 8.36$. For all intents and purposes this is a risky investment. Note that by mathematical convention mean and variance are often referenced as μ and σ^2. This is particularly the case whenever we use it to parametrize a Gaussian distribution.

In the same way as we introduced expectations and variance for *scalar* random variables, we can do so for vector-valued ones. Expectations are easy, since we can apply them elementwise. For instance, $\boldsymbol{\mu} \stackrel{\text{def}}{=} E_{\mathbf{x}\sim P}[\mathbf{x}]$ has coordinates $\mu_i = E_{\mathbf{x}\sim P}[x_i]$. *Covariances*

are more complicated. We define them by taking expectations of the *outer product* of the difference between random variables and their mean:

$$\boldsymbol{\Sigma} \stackrel{\text{def}}{=} \mathrm{Cov}_{\mathbf{x}\sim P}[\mathbf{x}] = E_{\mathbf{x}\sim P}\left[(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^\top\right]. \tag{2.6.16}$$

This matrix $\boldsymbol{\Sigma}$ is referred to as the covariance matrix. An easy way to see its effect is to consider some vector $\mathbf{v}$ of the same size as $\mathbf{x}$. It follows that

$$\mathbf{v}^\top\boldsymbol{\Sigma}\mathbf{v} = E_{\mathbf{x}\sim P}\left[\mathbf{v}^\top(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^\top\mathbf{v}\right] = \mathrm{Var}_{x\sim P}[\mathbf{v}^\top\mathbf{x}]. \tag{2.6.17}$$

As such, $\boldsymbol{\Sigma}$ allows us to compute the variance for any linear function of $\mathbf{x}$ by a simple matrix multiplication. The off-diagonal elements tell us how correlated the coordinates are: a value of 0 means no correlation, where a larger positive value means that they are more strongly correlated.

2.6.7 Discussion

In machine learning, there are many things to be uncertain about! We can be uncertain about the value of a label given an input. We can be uncertain about the estimated value of a parameter. We can even be uncertain about whether data arriving at deployment is even from the same distribution as the training data.

By *aleatoric uncertainty*, we mean uncertainty that is intrinsic to the problem, and due to genuine randomness unaccounted for by the observed variables. By *epistemic uncertainty*, we mean uncertainty over a model's parameters, the sort of uncertainty that we can hope to reduce by collecting more data. We might have epistemic uncertainty concerning the probability that a coin turns up heads, but even once we know this probability, we are left with aleatoric uncertainty about the outcome of any future toss. No matter how long we watch someone tossing a fair coin, we will never be more or less than 50% certain that the next toss will come up heads. These terms come from mechanical modeling, (see e.g., Der Kiureghian and Ditlevsen (2009) for a review on this aspect of uncertainty quantification[59]). It is worth noting, however, that these terms constitute a slight abuse of language. The term *epistemic* refers to anything concerning *knowledge* and thus, in the philosophical sense, all uncertainty is epistemic.

59 

We saw that sampling data from some unknown probability distribution can provide us with information that can be used to estimate the parameters of the data generating distribution. That said, the rate at which this is possible can be quite slow. In our coin tossing example (and many others) we can do no better than to design estimators that converge at a rate of $1/\sqrt{n}$, where n is the sample size (e.g., the number of tosses). This means that by going from 10 to 1000 observations (usually a very achievable task) we see a tenfold reduction of uncertainty, whereas the next 1000 observations help comparatively little, offering only a 1.41 times reduction. This is a persistent feature of machine learning: while there are often easy gains, it takes a very large amount of data, and often with it an enormous amount of computation, to make further gains. For an empirical review of this fact for large scale language models see Revels *et al.* (2016).

We also sharpened our language and tools for statistical modeling. In the process of that

we learned about conditional probabilities and about one of the most important equations in statistics—Bayes' theorem. It is an effective tool for decoupling information conveyed by data through a likelihood term $P(B \mid A)$ that addresses how well observations B match a choice of parameters A, and a prior probability $P(A)$ which governs how plausible a particular choice of A was in the first place. In particular, we saw how this rule can be applied to assign probabilities to diagnoses, based on the efficacy of the test *and* the prevalence of the disease itself (i.e., our prior).

Lastly, we introduced a first set of nontrivial questions about the effect of a specific probability distribution, namely expectations and variances. While there are many more than just linear and quadratic expectations for a probability distribution, these two already provide a good deal of knowledge about the possible behavior of the distribution. For instance, Chebyshev's inequality[60] states that $P(|X - \mu| \geq k\sigma) \leq 1/k^2$, where μ is the expectation, σ^2 is the variance of the distribution, and $k > 1$ is a confidence parameter of our choosing. It tells us that draws from a distribution lie with at least 50% probability within a $[-\sqrt{2}\sigma, \sqrt{2}\sigma]$ interval centered on the expectation.

60 

2.6.8 Exercises

1. Give an example where observing more data can reduce the amount of uncertainty about the outcome to an arbitrarily low level.
2. Give an example where observing more data will only reduce the amount of uncertainty up to a point and then no further. Explain why this is the case and where you expect this point to occur.
3. We empirically demonstrated convergence to the mean for the toss of a coin. Calculate the variance of the estimate of the probability that we see a head after drawing n samples.
   1. How does the variance scale with the number of observations?
   2. Use Chebyshev's inequality to bound the deviation from the expectation.
   3. How does it relate to the central limit theorem?
4. Assume that we draw m samples x_i from a probability distribution with zero mean and unit variance. Compute the averages $z_m \stackrel{\text{def}}{=} m^{-1} \sum_{i=1}^{m} x_i$. Can we apply Chebyshev's inequality for every z_m independently? Why not?
5. Given two events with probability $P(\mathcal{A})$ and $P(\mathcal{B})$, compute upper and lower bounds on $P(\mathcal{A} \cup \mathcal{B})$ and $P(\mathcal{A} \cap \mathcal{B})$. Hint: graph the situation using a Venn diagram[61].

61 

6. Assume that we have a sequence of random variables, say A, B, and C, where B only depends on A, and C only depends on B, can you simplify the joint probability $P(A, B, C)$? Hint: this is a Markov chain[62].

62 

7. In Section 2.6.5, assume that the outcomes of the two tests are not independent. In particular assume that either test on its own has a false positive rate of 10% and a false negative rate of 1%. That is, assume that $P(D = 1 \mid H = 0) = 0.1$ and that $P(D = 0 \mid H = 1) = 0.01$. Moreover, assume that for $H = 1$ (infected) the test outcomes are

conditionally independent, i.e., that $P(D_1, D_2 \mid H = 1) = P(D_1 \mid H = 1)P(D_2 \mid H = 1)$ but that for healthy patients the outcomes are coupled via $P(D_1 = D_2 = 1 \mid H = 0) = 0.02$.

1. Work out the joint probability table for D_1 and D_2, given $H = 0$ based on the information you have so far.
2. Derive the probability that the patient is diseased ($H = 1$) after one test returns positive. You can assume the same baseline probability $P(H = 1) = 0.0015$ as before.
3. Derive the probability that the patient is diseased ($H = 1$) after both tests return positive.

8. Assume that you are an asset manager for an investment bank and you have a choice of stocks s_i to invest in. Your portfolio needs to add up to 1 with weights α_i for each stock. The stocks have an average return $\boldsymbol{\mu} = E_{\mathbf{s}\sim P}[\mathbf{s}]$ and covariance $\boldsymbol{\Sigma} = \mathrm{Cov}_{\mathbf{s}\sim P}[\mathbf{s}]$.
   1. Compute the expected return for a given portfolio $\boldsymbol{\alpha}$.
   2. If you wanted to maximize the return of the portfolio, how should you choose your investment?
   3. Compute the *variance* of the portfolio.
   4. Formulate an optimization problem of maximizing the return while keeping the variance constrained to an upper bound. This is the Nobel-Prize winning Markovitz portfolio[63] (Mangram, 2013). To solve it you will need a quadratic programming solver, something way beyond the scope of this book.

[63] 

Discussions[64].

[64] 

2.7 Documentation

While we cannot possibly introduce every single PyTorch function and class (and the information might become outdated quickly), the API documentation[65] and additional tutorials [66] and examples provide such documentation. This section provides some guidance for how to explore the PyTorch API.

[65] 

[66] 

```
import torch
```

2.7.1 Functions and Classes in a Module

To know which functions and classes can be called in a module, we invoke the `dir` function. For instance, we can query all properties in the module for generating random numbers:

```
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']
```

Generally, we can ignore functions that start and end with __ (special objects in Python) or functions that start with a single _(usually internal functions). Based on the remaining function or attribute names, we might hazard a guess that this module offers various methods for generating random numbers, including sampling from the uniform distribution (`uniform`), normal distribution (`normal`), and multinomial distribution (`multinomial`).

2.7.2 Specific Functions and Classes

For specific instructions on how to use a given function or class, we can invoke the `help` function. As an example, let's explore the usage instructions for tensors' `ones` function.

```
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.])
```

From the documentation, we can see that the ones function creates a new tensor with the specified shape and sets all the elements to the value of 1. Whenever possible, you should run a quick test to confirm your interpretation:

```
torch.ones(4)
```

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

In the Jupyter notebook, we can use ? to display the document in another window. For example, `list?` will create content that is almost identical to `help(list)`, displaying it in a new browser window. In addition, if we use two question marks, such as `list??`, the Python code implementing the function will also be displayed.

The official documentation provides plenty of descriptions and examples that are beyond this book. We emphasize important use cases that will get you started quickly with practical problems, rather than completeness of coverage. We also encourage you to study the source code of the libraries to see examples of high-quality implementations of production code. By doing this you will become a better engineer in addition to becoming a better scientist.

Discussions[67].

67 

3 Linear Neural Networks for Regression

Before we worry about making our neural networks deep, it will be helpful to implement some shallow ones, for which the inputs connect directly to the outputs. This will prove important for a few reasons. First, rather than getting distracted by complicated architectures, we can focus on the basics of neural network training, including parametrizing the output layer, handling data, specifying a loss function, and training the model. Second, this class of shallow networks happens to comprise the set of linear models, which subsumes many classical methods of statistical prediction, including linear and softmax regression. Understanding these classical tools is pivotal because they are widely used in many contexts and we will often need to use them as baselines when justifying the use of fancier architectures. This chapter will focus narrowly on linear regression and the next one will extend our modeling repertoire by developing linear neural networks for classification.

3.1 Linear Regression

Regression problems pop up whenever we want to predict a numerical value. Common examples include predicting prices (of homes, stocks, etc.), predicting the length of stay (for patients in the hospital), forecasting demand (for retail sales), among numerous others. Not every prediction problem is one of classical regression. Later on, we will introduce classification problems, where the goal is to predict membership among a set of categories.

As a running example, suppose that we wish to estimate the prices of houses (in dollars) based on their area (in square feet) and age (in years). To develop a model for predicting house prices, we need to get our hands on data, including the sales price, area, and age for each home. In the terminology of machine learning, the dataset is called a *training dataset* or *training set*, and each row (containing the data corresponding to one sale) is called an *example* (or *data point*, *instance*, *sample*). The thing we are trying to predict (price) is called a *label* (or *target*). The variables (age and area) upon which the predictions are based are called *features* (or *covariates*).

```
%matplotlib inline
import math
import time
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import torch
from d2l import torch as d2l
```

3.1.1 Basics

Linear regression is both the simplest and most popular among the standard tools for tackling regression problems. Dating back to the dawn of the 19th century (Gauss, 1809, Legendre, 1805), linear regression flows from a few simple assumptions. First, we assume that the relationship between features $\mathbf{x}$ and target y is approximately linear, i.e., that the conditional mean $E[Y \mid X = \mathbf{x}]$ can be expressed as a weighted sum of the features $\mathbf{x}$. This setup allows that the target value may still deviate from its expected value on account of observation noise. Next, we can impose the assumption that any such noise is well behaved, following a Gaussian distribution. Typically, we will use n to denote the number of examples in our dataset. We use superscripts to enumerate samples and targets, and subscripts to index coordinates. More concretely, $\mathbf{x}^{(i)}$ denotes the i^{th} sample and $x_j^{(i)}$ denotes its j^{th} coordinate.

Model

At the heart of every solution is a model that describes how features can be transformed into an estimate of the target. The assumption of linearity means that the expected value of the target (price) can be expressed as a weighted sum of the features (area and age):

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b. \tag{3.1.1}$$

Here w_{area} and w_{age} are called *weights*, and b is called a *bias* (or *offset* or *intercept*). The weights determine the influence of each feature on our prediction. The bias determines the value of the estimate when all features are zero. Even though we will never see any newly-built homes with precisely zero area, we still need the bias because it allows us to express all linear functions of our features (rather than restricting us to lines that pass through the origin). Strictly speaking, (3.1.1) is an *affine transformation* of input features, which is characterized by a *linear transformation* of features via a weighted sum, combined with a *translation* via the added bias. Given a dataset, our goal is to choose the weights $\mathbf{w}$ and the bias b that, on average, make our model's predictions fit the true prices observed in the data as closely as possible.

In disciplines where it is common to focus on datasets with just a few features, explicitly expressing models long-form, as in (3.1.1), is common. In machine learning, we usually work with high-dimensional datasets, where it is more convenient to employ compact linear algebra notation. When our inputs consist of d features, we can assign each an index (between 1 and d) and express our prediction $\hat{y}$ (in general the "hat" symbol denotes an estimate) as

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b. \tag{3.1.2}$$

Collecting all features into a vector $\mathbf{x} \in \mathbb{R}^d$ and all weights into a vector $\mathbf{w} \in \mathbb{R}^d$, we can express our model compactly via the dot product between $\mathbf{w}$ and $\mathbf{x}$:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b. \tag{3.1.3}$$

In (3.1.3), the vector $\mathbf{x}$ corresponds to the features of a single example. We will often find it convenient to refer to features of our entire dataset of n examples via the *design matrix* $\mathbf{X} \in \mathbb{R}^{n \times d}$. Here, $\mathbf{X}$ contains one row for every example and one column for every feature. For a collection of features $\mathbf{X}$, the predictions $\hat{\mathbf{y}} \in \mathbb{R}^n$ can be expressed via the matrix–vector product:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b, \tag{3.1.4}$$

where broadcasting (Section 2.1.4) is applied during the summation. Given features of a training dataset $\mathbf{X}$ and corresponding (known) labels $\mathbf{y}$, the goal of linear regression is to find the weight vector $\mathbf{w}$ and the bias term b such that, given features of a new data example sampled from the same distribution as $\mathbf{X}$, the new example's label will (in expectation) be predicted with the smallest error.

Even if we believe that the best model for predicting y given $\mathbf{x}$ is linear, we would not expect to find a real-world dataset of n examples where $y^{(i)}$ exactly equals $\mathbf{w}^\top \mathbf{x}^{(i)} + b$ for all $1 \leq i \leq n$. For example, whatever instruments we use to observe the features $\mathbf{X}$ and labels $\mathbf{y}$, there might be a small amount of measurement error. Thus, even when we are confident that the underlying relationship is linear, we will incorporate a noise term to account for such errors.

Before we can go about searching for the best *parameters* (or *model parameters*) $\mathbf{w}$ and b, we will need two more things: (i) a measure of the quality of some given model; and (ii) a procedure for updating the model to improve its quality.

Loss Function

Naturally, fitting our model to the data requires that we agree on some measure of *fitness* (or, equivalently, of *unfitness*). *Loss functions* quantify the distance between the *real* and *predicted* values of the target. The loss will usually be a nonnegative number where smaller values are better and perfect predictions incur a loss of 0. For regression problems, the most common loss function is the squared error. When our prediction for an example i is $\hat{y}^{(i)}$ and the corresponding true label is $y^{(i)}$, the *squared error* is given by:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{3.1.5}$$

The constant $\frac{1}{2}$ makes no real difference but proves to be notationally convenient, since it cancels out when we take the derivative of the loss. Because the training dataset is given to us, and thus is out of our control, the empirical error is only a function of the model parameters. In Fig. 3.1.1, we visualize the fit of a linear regression model in a problem with one-dimensional inputs.

Note that large differences between estimates $\hat{y}^{(i)}$ and targets $y^{(i)}$ lead to even larger contributions to the loss, due to its quadratic form (this quadraticity can be a double-edge sword;

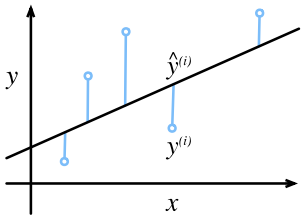


Fig. 3.1.1 Fitting a linear regression model to one-dimensional data.

while it encourages the model to avoid large errors it can also lead to excessive sensitivity to anomalous data). To measure the quality of a model on the entire dataset of n examples, we simply average (or equivalently, sum) the losses on the training set:

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{3.1.6}$$

When training the model, we seek parameters $(\mathbf{w}^*, b^*)$ that minimize the total loss across all training examples:

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}}\ L(\mathbf{w}, b). \tag{3.1.7}$$

Analytic Solution

Unlike most of the models that we will cover, linear regression presents us with a surprisingly easy optimization problem. In particular, we can find the optimal parameters (as assessed on the training data) analytically by applying a simple formula as follows. First, we can subsume the bias b into the parameter $\mathbf{w}$ by appending a column to the design matrix consisting of all 1s. Then our prediction problem is to minimize $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$. As long as the design matrix $\mathbf{X}$ has full rank (no feature is linearly dependent on the others), then there will be just one critical point on the loss surface and it corresponds to the minimum of the loss over the entire domain. Taking the derivative of the loss with respect to $\mathbf{w}$ and setting it equal to zero yields:

$$\partial_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 = 2\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \text{ and hence } \mathbf{X}^\top \mathbf{y} = \mathbf{X}^\top \mathbf{X}\mathbf{w}. \tag{3.1.8}$$

Solving for $\mathbf{w}$ provides us with the optimal solution for the optimization problem. Note that this solution

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y} \tag{3.1.9}$$

will only be unique when the matrix $\mathbf{X}^\top \mathbf{X}$ is invertible, i.e., when the columns of the design matrix are linearly independent (Golub and Van Loan, 1996).

While simple problems like linear regression may admit analytic solutions, you should not get used to such good fortune. Although analytic solutions allow for nice mathematical analysis, the requirement of an analytic solution is so restrictive that it would exclude almost all exciting aspects of deep learning.

Minibatch Stochastic Gradient Descent

Fortunately, even in cases where we cannot solve the models analytically, we can still often train models effectively in practice. Moreover, for many tasks, those hard-to-optimize models turn out to be so much better that figuring out how to train them ends up being well worth the trouble.

The key technique for optimizing nearly every deep learning model, and which we will call upon throughout this book, consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function. This algorithm is called *gradient descent*.

The most naive application of gradient descent consists of taking the derivative of the loss function, which is an average of the losses computed on every single example in the dataset. In practice, this can be extremely slow: we must pass over the entire dataset before making a single update, even if the update steps might be very powerful (Liu and Nocedal, 1989). Even worse, if there is a lot of redundancy in the training data, the benefit of a full update is limited.

The other extreme is to consider only a single example at a time and to take update steps based on one observation at a time. The resulting algorithm, *stochastic gradient descent* (SGD) can be an effective strategy (Bottou, 2010), even for large datasets. Unfortunately, SGD has drawbacks, both computational and statistical. One problem arises from the fact that processors are a lot faster multiplying and adding numbers than they are at moving data from main memory to processor cache. It is up to an order of magnitude more efficient to perform a matrix–vector multiplication than a corresponding number of vector–vector operations. This means that it can take a lot longer to process one sample at a time compared to a full batch. A second problem is that some of the layers, such as batch normalization (to be described in Section 8.5), only work well when we have access to more than one observation at a time.

The solution to both problems is to pick an intermediate strategy: rather than taking a full batch or only a single sample at a time, we take a *minibatch* of observations (Li *et al.*, 2014). The specific choice of the size of the said minibatch depends on many factors, such as the amount of memory, the number of accelerators, the choice of layers, and the total dataset size. Despite all that, a number between 32 and 256, preferably a multiple of a large power of 2, is a good start. This leads us to *minibatch stochastic gradient descent*.

In its most basic form, in each iteration t, we first randomly sample a minibatch $\mathcal{B}_t$ consisting of a fixed number $|\mathcal{B}|$ of training examples. We then compute the derivative (gradient) of the average loss on the minibatch with respect to the model parameters. Finally, we multiply the gradient by a predetermined small positive value η, called the *learning rate*, and subtract the resulting term from the current parameter values. We can express the update as follows:

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b). \tag{3.1.10}$$

In summary, minibatch SGD proceeds as follows: (i) initialize the values of the model

parameters, typically at random; (ii) iteratively sample random minibatches from the data, updating the parameters in the direction of the negative gradient. For quadratic losses and affine transformations, this has a closed-form expansion:

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) &&= \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right) \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_b l^{(i)}(\mathbf{w}, b) &&= b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right). \end{aligned} \tag{3.1.11}$$

Since we pick a minibatch $\mathcal{B}$ we need to normalize by its size $|\mathcal{B}|$. Frequently minibatch size and learning rate are user-defined. Such tunable parameters that are not updated in the training loop are called *hyperparameters*. They can be tuned automatically by a number of techniques, such as Bayesian optimization (Frazier, 2018). In the end, the quality of the solution is typically assessed on a separate *validation dataset* (or *validation set*).

After training for some predetermined number of iterations (or until some other stopping criterion is met), we record the estimated model parameters, denoted $\hat{\mathbf{w}}, \hat{b}$. Note that even if our function is truly linear and noiseless, these parameters will not be the exact minimizers of the loss, nor even deterministic. Although the algorithm converges slowly towards the minimizers it typically will not find them exactly in a finite number of steps. Moreover, the minibatches $\mathcal{B}$ used for updating the parameters are chosen at random. This breaks determinism.

Linear regression happens to be a learning problem with a global minimum (whenever $\mathbf{X}$ is full rank, or equivalently, whenever $\mathbf{X}^\top \mathbf{X}$ is invertible). However, the loss surfaces for deep networks contain many saddle points and minima. Fortunately, we typically do not care about finding an exact set of parameters but merely any set of parameters that leads to accurate predictions (and thus low loss). In practice, deep learning practitioners seldom struggle to find parameters that minimize the loss *on training sets* (Frankle and Carbin, 2018, Izmailov *et al.*, 2018). The more formidable task is to find parameters that lead to accurate predictions on previously unseen data, a challenge called *generalization*. We return to these topics throughout the book.

Predictions

Given the model $\hat{\mathbf{w}}^\top \mathbf{x} + \hat{b}$, we can now make *predictions* for a new example, e.g., predicting the sales price of a previously unseen house given its area x_1 and age x_2. Deep learning practitioners have taken to calling the prediction phase *inference* but this is a bit of a misnomer—*inference* refers broadly to any conclusion reached on the basis of evidence, including both the values of the parameters and the likely label for an unseen instance. If anything, in the statistics literature *inference* more often denotes parameter inference and this overloading of terminology creates unnecessary confusion when deep learning practitioners talk to statisticians. In the following we will stick to *prediction* whenever possible.

3.1.2 Vectorization for Speed

When training our models, we typically want to process whole minibatches of examples simultaneously. Doing this efficiently requires that we vectorize the calculations and leverage fast linear algebra libraries rather than writing costly for-loops in Python.

To see why this matters so much, let's consider two methods for adding vectors. To start, we instantiate two 10,000-dimensional vectors containing all 1s. In the first method, we loop over the vectors with a Python for-loop. In the second, we rely on a single call to +.

```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

Now we can benchmark the workloads. First, we add them, one coordinate at a time, using a for-loop.

```
c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

```
'0.17802 sec'
```

Alternatively, we rely on the reloaded + operator to compute the elementwise sum.

```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

```
'0.00036 sec'
```

The second method is dramatically faster than the first. Vectorizing code often yields order-of-magnitude speedups. Moreover, we push more of the mathematics to the library so we do not have to write as many calculations ourselves, reducing the potential for errors and increasing portability of the code.

3.1.3 The Normal Distribution and Squared Loss

So far we have given a fairly functional motivation of the squared loss objective: the optimal parameters return the conditional expectation $E[Y \mid X]$ whenever the underlying pattern is truly linear, and the loss assigns large penalties for outliers. We can also provide a more formal motivation for the squared loss objective by making probabilistic assumptions about the distribution of noise.

Linear regression was invented at the turn of the 19th century. While it has long been debated whether Gauss or Legendre first thought up the idea, it was Gauss who also discovered the normal distribution (also called the *Gaussian*). It turns out that the normal

distribution and linear regression with squared loss share a deeper connection than common parentage.

To begin, recall that a normal distribution with mean μ and variance σ^2 (standard deviation σ) is given as

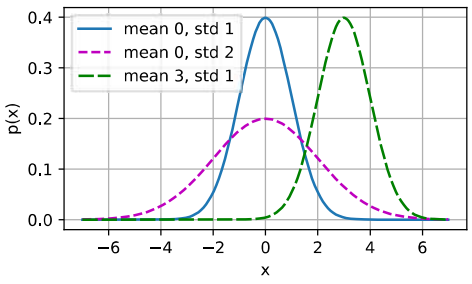
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right). \tag{3.1.12}$$

Below we define a function to compute the normal distribution.

```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 * (x - mu)**2 / sigma**2)
```

We can now visualize the normal distributions.

```
# 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])
```



Note that changing the mean corresponds to a shift along the x-axis, and increasing the variance spreads the distribution out, lowering its peak.

One way to motivate linear regression with squared loss is to assume that observations arise from noisy measurements, where the noise ϵ follows the normal distribution $\mathcal{N}(0, \sigma^2)$:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{3.1.13}$$

Thus, we can now write out the *likelihood* of seeing a particular y for a given $\mathbf{x}$ via

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \tag{3.1.14}$$

As such, the likelihood factorizes. According to *the principle of maximum likelihood*, the

best values of parameters $\mathbf{w}$ and b are those that maximize the *likelihood* of the entire dataset:

$$P(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} p(y^{(i)} \mid \mathbf{x}^{(i)}). \tag{3.1.15}$$

The equality follows since all pairs $(\mathbf{x}^{(i)}, y^{(i)})$ were drawn independently of each other. Estimators chosen according to the principle of maximum likelihood are called *maximum likelihood estimators*. While, maximizing the product of many exponential functions, might look difficult, we can simplify things significantly, without changing the objective, by maximizing the logarithm of the likelihood instead. For historical reasons, optimizations are more often expressed as minimization rather than maximization. So, without changing anything, we can *minimize* the *negative log-likelihood*, which we can express as follows:

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \tag{3.1.16}$$

If we assume that σ is fixed, we can ignore the first term, because it does not depend on $\mathbf{w}$ or b. The second term is identical to the squared error loss introduced earlier, except for the multiplicative constant $\frac{1}{\sigma^2}$. Fortunately, the solution does not depend on σ either. It follows that minimizing the mean squared error is equivalent to the maximum likelihood estimation of a linear model under the assumption of additive Gaussian noise.

3.1.4 Linear Regression as a Neural Network

While linear models are not sufficiently rich to express the many complicated networks that we will introduce in this book, (artificial) neural networks are rich enough to subsume linear models as networks in which every feature is represented by an input neuron, all of which are connected directly to the output.

Fig. 3.1.2 depicts linear regression as a neural network. The diagram highlights the connectivity pattern, such as how each input is connected to the output, but not the specific values taken by the weights or biases.

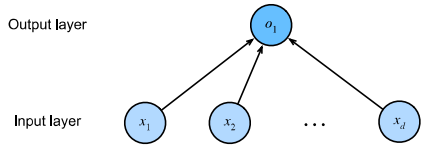


Fig. 3.1.2 Linear regression is a single-layer neural network.

The inputs are $x_1, \ldots, x_d$. We refer to d as the *number of inputs* or the *feature dimensionality* in the input layer. The output of the network is o_1. Because we are just trying to predict a single numerical value, we have only one output neuron. Note that the input values are all *given*. There is just a single *computed* neuron. In summary, we can think of linear regression as a single-layer fully connected neural network. We will encounter networks with far more layers in later chapters.

Biology

Because linear regression predates computational neuroscience, it might seem anachronistic to describe linear regression in terms of neural networks. Nonetheless, they were a natural place to start when the cyberneticists and neurophysiologists Warren McCulloch and Walter Pitts began to develop models of artificial neurons. Consider the cartoonish picture of a biological neuron in Fig. 3.1.3, consisting of *dendrites* (input terminals), the *nucleus* (CPU), the *axon* (output wire), and the *axon terminals* (output terminals), enabling connections to other neurons via *synapses*.

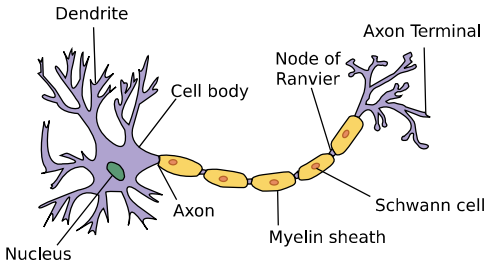


Fig. 3.1.3 The real neuron (source: "Anatomy and Physiology" by the US National Cancer Institute's Surveillance, Epidemiology and End Results (SEER) Program).

Information x_i arriving from other neurons (or environmental sensors) is received in the dendrites. In particular, that information is weighted by *synaptic weights* w_i, determining the effect of the inputs, e.g., activation or inhibition via the product $x_i w_i$. The weighted inputs arriving from multiple sources are aggregated in the nucleus as a weighted sum $y = \sum_i x_i w_i + b$, possibly subject to some nonlinear postprocessing via a function $\sigma(y)$. This information is then sent via the axon to the axon terminals, where it reaches its destination (e.g., an actuator such as a muscle) or it is fed into another neuron via its dendrites.

Certainly, the high-level idea that many such units could be combined, provided they have the correct connectivity and learning algorithm, to produce far more interesting and complex behavior than any one neuron alone could express arises from our study of real biological neural systems. At the same time, most research in deep learning today draws inspiration from a much wider source. We invoke Russell and Norvig (2016) who pointed out that although airplanes might have been *inspired* by birds, ornithology has not been the primary driver of aeronautics innovation for some centuries. Likewise, inspiration in deep learning these days comes in equal or greater measure from mathematics, linguistics, psychology, statistics, computer science, and many other fields.

3.1.5 Summary

In this section, we introduced traditional linear regression, where the parameters of a linear function are chosen to minimize squared loss on the training set. We also motivated this choice of objective both via some practical considerations and through an interpretation of linear regression as maximimum likelihood estimation under an assumption of linearity and Gaussian noise. After discussing both computational considerations and connections to

statistics, we showed how such linear models could be expressed as simple neural networks where the inputs are directly wired to the output(s). While we will soon move past linear models altogether, they are sufficient to introduce most of the components that all of our models require: parametric forms, differentiable objectives, optimization via minibatch stochastic gradient descent, and ultimately, evaluation on previously unseen data.

3.1.6 Exercises

1. Assume that we have some data $x_1, \ldots, x_n \in \mathbb{R}$. Our goal is to find a constant b such that $\sum_i (x_i - b)^2$ is minimized.
   1. Find an analytic solution for the optimal value of b.
   2. How does this problem and its solution relate to the normal distribution?
   3. What if we change the loss from $\sum_i (x_i - b)^2$ to $\sum_i |x_i - b|$? Can you find the optimal solution for b?
2. Prove that the affine functions that can be expressed by $\mathbf{x}^\top \mathbf{w} + b$ are equivalent to linear functions on $(\mathbf{x}, 1)$.
3. Assume that you want to find quadratic functions of $\mathbf{x}$, i.e., $f(\mathbf{x}) = b + \sum_i w_i x_i + \sum_{j \leq i} w_{ij} x_i x_j$. How would you formulate this in a deep network?
4. Recall that one of the conditions for the linear regression problem to be solvable was that the design matrix $\mathbf{X}^\top \mathbf{X}$ has full rank.
   1. What happens if this is not the case?
   2. How could you fix it? What happens if you add a small amount of coordinate-wise independent Gaussian noise to all entries of $\mathbf{X}$?
   3. What is the expected value of the design matrix $\mathbf{X}^\top \mathbf{X}$ in this case?
   4. What happens with stochastic gradient descent when $\mathbf{X}^\top \mathbf{X}$ does not have full rank?
5. Assume that the noise model governing the additive noise ϵ is the exponential distribution. That is, $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$.
   1. Write out the negative log-likelihood of the data under the model $-\log P(\mathbf{y} \mid \mathbf{X})$.
   2. Can you find a closed form solution?
   3. Suggest a minibatch stochastic gradient descent algorithm to solve this problem. What could possibly go wrong (hint: what happens near the stationary point as we keep on updating the parameters)? Can you fix this?
6. Assume that we want to design a neural network with two layers by composing two linear layers. That is, the output of the first layer becomes the input of the second layer. Why would such a naive composition not work?
7. What happens if you want to use regression for realistic price estimation of houses or stock prices?

1. Show that the additive Gaussian noise assumption is not appropriate. Hint: can we have negative prices? What about fluctuations?
2. Why would regression to the logarithm of the price be much better, i.e., $y = \log \text{price}$?
3. What do you need to worry about when dealing with pennystock, i.e., stock with very low prices? Hint: can you trade at all possible prices? Why is this a bigger problem for cheap stock? For more information review the celebrated Black–Scholes model for option pricing (Black and Scholes, 1973).

8. Suppose we want to use regression to estimate the *number* of apples sold in a grocery store.
   1. What are the problems with a Gaussian additive noise model? Hint: you are selling apples, not oil.
   2. The Poisson distribution[68] captures distributions over counts. It is given by $p(k \mid \lambda) = \lambda^k e^{-\lambda}/k!$. Here λ is the rate function and k is the number of events you see. Prove that λ is the expected value of counts k.
   3. Design a loss function associated with the Poisson distribution.
   4. Design a loss function for estimating $\log \lambda$ instead.

[68] 

[69]  Discussions[69].

3.2 Object-Oriented Design for Implementation

In our introduction to linear regression, we walked through various components including the data, the model, the loss function, and the optimization algorithm. Indeed, linear regression is one of the simplest machine learning models. Training it, however, uses many of the same components that other models in this book require. Therefore, before diving into the implementation details it is worth designing some of the APIs that we use throughout. Treating components in deep learning as objects, we can start by defining classes for these objects and their interactions. This object-oriented design for implementation will greatly streamline the presentation and you might even want to use it in your projects.

[70] 

Inspired by open-source libraries such as PyTorch Lightning[70], at a high level we wish to have three classes: (i) `Module` contains models, losses, and optimization methods; (ii) `DataModule` provides data loaders for training and validation; (iii) both classes are combined using the `Trainer` class, which allows us to train models on a variety of hardware platforms. Most code in this book adapts `Module` and `DataModule`. We will touch upon the `Trainer` class only when we discuss GPUs, CPUs, parallel training, and optimization algorithms.

```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

3.2.1 Utilities

We need a few utilities to simplify object-oriented programming in Jupyter notebooks. One of the challenges is that class definitions tend to be fairly long blocks of code. Notebook readability demands short code fragments, interspersed with explanations, a requirement incompatible with the style of programming common for Python libraries. The first utility function allows us to register functions as methods in a class *after* the class has been created. In fact, we can do so *even after* we have created instances of the class! It allows us to split the implementation of a class into multiple code blocks.

```
def add_to_class(Class):  #@save
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

Let's have a quick look at how to use it. We plan to implement a class `A` with a method `do`. Instead of having code for both `A` and `do` in the same code block, we can first declare the class `A` and create an instance `a`.

```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

Next we define the method `do` as we normally would, but not in class `A`'s scope. Instead, we decorate this method by `add_to_class` with class `A` as its argument. In doing so, the method is able to access the member variables of `A` just as we would expect had it been included as part of `A`'s definition. Let's see what happens when we invoke it for the instance `a`.

```
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a.do()
```

```
Class attribute "b" is 1
```

The second one is a utility class that saves all arguments in a class's `__init__` method

as class attributes. This allows us to extend constructor call signatures implicitly without additional code.

```
class HyperParameters:  #@save
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

We defer its implementation into Section B.7. To use it, we define our class that inherits from `HyperParameters` and calls `save_hyperparameters` in the `__init__` method.

```
# 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
```

The final utility allows us to plot experiment progress interactively while it is going on. In deference to the much more powerful (and complex) TensorBoard[71] we name it `ProgressBoard`. The implementation is deferred to Section B.7. For now, let's simply see it in action.

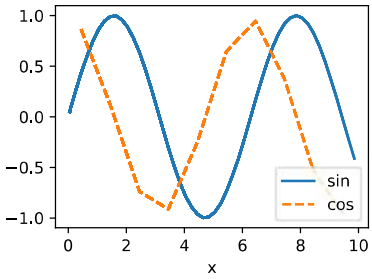
71 

The `draw` method plots a point `(x, y)` in the figure, with `label` specified in the legend. The optional `every_n` smooths the line by only showing $1/n$ points in the figure. Their values are averaged from the n neighbor points in the original figure.

```
class ProgressBoard(d2l.HyperParameters):  #@save
    """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
```

In the following example, we draw `sin` and `cos` with a different smoothness. If you run this code block, you will see the lines grow in animation.

```
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)
```



3.2.2 Models

The `Module` class is the base class of all models we will implement. At the very least we need three methods. The first, `__init__`, stores the learnable parameters, the `training_step` method accepts a data batch to return the loss value, and finally, `configure_optimizers` returns the optimization method, or a list of them, that is used to update the learnable parameters. Optionally we can define `validation_step` to report the evaluation measures. Sometimes we put the code for computing the output into a separate `forward` method to make it more reusable.

```
class Module(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):
        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, value.to(d2l.cpu()).detach().numpy(),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))
```

(continues on next page)

(continued from previous page)

```
    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
```

You may notice that `Module` is a subclass of `nn.Module`, the base class of neural networks in PyTorch. It provides convenient features for handling neural networks. For example, if we define a `forward` method, such as `forward(self, X)`, then for an instance `a` we can invoke this method by `a(X)`. This works since it calls the `forward` method in the built-in `__call__` method. You can find more details and examples about `nn.Module` in Section 6.1.

3.2.3 Data

The `DataModule` class is the base class for data. Quite frequently the `__init__` method is used to prepare the data. This includes downloading and preprocessing if needed. The `train_dataloader` returns the data loader for the training dataset. A data loader is a (Python) generator that yields a data batch each time it is used. This batch is then fed into the `training_step` method of `Module` to compute the loss. There is an optional `val_dataloader` to return the validation dataset loader. It behaves in the same manner, except that it yields data batches for the `validation_step` method in `Module`.

```
class DataModule(d2l.HyperParameters):  #@save
    """The base class of data."""
    def __init__(self, root='../data', num_workers=4):
        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)
```

3.2.4 Training

The `Trainer` class trains the learnable parameters in the `Module` class with data specified in `DataModule`. The key method is `fit`, which accepts two arguments: `model`, an instance of `Module`, and `data`, an instance of `DataModule`. It then iterates over the entire dataset

`max_epochs` times to train the model. As before, we will defer the implementation of this method to later chapters.

```
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

    def fit(self, model, data):
        self.prepare_data(data)
        self.prepare_model(model)
        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()

    def fit_epoch(self):
        raise NotImplementedError
```

3.2.5 Summary

To highlight the object-oriented design for our future deep learning implementation, the above classes simply show how their objects store data and interact with each other. We will keep enriching implementations of these classes, such as via `@add_to_class`, in the rest of the book. Moreover, these fully implemented classes are saved in the D2L library[72]  , a *lightweight toolkit* that makes structured modeling for deep learning easy. In particular, it facilitates reusing many components between projects without changing much at all. For instance, we can replace just the optimizer, just the model, just the dataset, etc.; this degree of modularity pays dividends throughout the book in terms of conciseness and simplicity (this is why we added it) and it can do the same for your own projects.

3.2.6 Exercises

1. Locate full implementations of the above classes that are saved in the D2L library[73]  . We strongly recommend that you look at the implementation in detail once you have gained some more familiarity with deep learning modeling.

2. Remove the `save_hyperparameters` statement in the `B` class. Can you still print `self.a` and `self.b`? Optional: if you have dived into the full implementation of the `HyperParameters` class, can you explain why?

Discussions[74].

[74] 

3.3 Synthetic Regression Data

Machine learning is all about extracting information from data. So you might wonder, what could we possibly learn from synthetic data? While we might not care intrinsically about the patterns that we ourselves baked into an artificial data generating model, such datasets are nevertheless useful for didactic purposes, helping us to evaluate the properties of our learning algorithms and to confirm that our implementations work as expected. For example, if we create data for which the correct parameters are known *a priori*, then we can check that our model can in fact recover them.

```
%matplotlib inline
import random
import torch
from d2l import torch as d2l
```

3.3.1 Generating the Dataset

For this example, we will work in low dimension for succinctness. The following code snippet generates 1000 examples with 2-dimensional features drawn from a standard normal distribution. The resulting design matrix $\mathbf{X}$ belongs to $\mathbb{R}^{1000\times 2}$. We generate each label by applying a *ground truth* linear function, corrupting them via additive noise $\boldsymbol{\epsilon}$, drawn independently and identically for each example:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \boldsymbol{\epsilon}. \tag{3.3.1}$$

For convenience we assume that $\boldsymbol{\epsilon}$ is drawn from a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 0.01$. Note that for object-oriented design we add the code to the `__init__` method of a subclass of `d2l.DataModule` (introduced in Section 3.2.3). It is good practice to allow the setting of any additional hyperparameters. We accomplish this with `save_hyperparameters()`. The `batch_size` will be determined later.

```
class SyntheticRegressionData(d2l.DataModule):  #@save
    """Synthetic data for linear regression."""
    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
```

(continues on next page)

(continued from previous page)

```
        self.X = torch.randn(n, len(w))
        noise = torch.randn(n, 1) * noise
        self.y = torch.matmul(self.X, w.reshape((-1, 1))) + b + noise
```

Below, we set the true parameters to $\mathbf{w} = [2, -3.4]^\top$ and $b = 4.2$. Later, we can check our estimated parameters against these *ground truth* values.

```
data = SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
```

Each row in `features` consists of a vector in $\mathbb{R}^2$ and each row in `labels` is a scalar. Let's have a look at the first entry.

```
print('features:', data.X[0],'\nlabel:', data.y[0])
```

```
features: tensor([0.9026, 1.0264])
label: tensor([2.5148])
```

3.3.2 Reading the Dataset

Training machine learning models often requires multiple passes over a dataset, grabbing one minibatch of examples at a time. This data is then used to update the model. To illustrate how this works, we implement the `get_dataloader` method, registering it in the `SyntheticRegressionData` class via `add_to_class` (introduced in Section 3.2.1). It takes a batch size, a matrix of features, and a vector of labels, and generates minibatches of size `batch_size`. As such, each minibatch consists of a tuple of features and labels. Note that we need to be mindful of whether we're in training or validation mode: in the former, we will want to read the data in random order, whereas for the latter, being able to read data in a pre-defined order may be important for debugging purposes.

```
@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):
        batch_indices = torch.tensor(indices[i: i+self.batch_size])
        yield self.X[batch_indices], self.y[batch_indices]
```

To build some intuition, let's inspect the first minibatch of data. Each minibatch of features provides us with both its size and the dimensionality of input features. Likewise, our minibatch of labels will have a matching shape given by `batch_size`.

```
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])
```

While seemingly innocuous, the invocation of `iter(data.train_dataloader())` illustrates the power of Python's object-oriented design. Note that we added a method to the `SyntheticRegressionData` class *after* creating the `data` object. Nonetheless, the object benefits from the *ex post facto* addition of functionality to the class.

Throughout the iteration we obtain distinct minibatches until the entire dataset has been exhausted (try this). While the iteration implemented above is good for didactic purposes, it is inefficient in ways that might get us into trouble with real problems. For example, it requires that we load all the data in memory and that we perform lots of random memory access. The built-in iterators implemented in a deep learning framework are considerably more efficient and they can deal with sources such as data stored in files, data received via a stream, and data generated or processed on the fly. Next let's try to implement the same method using built-in iterators.

3.3.3 Concise Implementation of the Data Loader

Rather than writing our own iterator, we can call the existing API in a framework to load data. As before, we need a dataset with features `X` and labels `y`. Beyond that, we set `batch_size` in the built-in data loader and let it take care of shuffling examples efficiently.

```
@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)
    dataset = torch.utils.data.TensorDataset(*tensors)
    return torch.utils.data.DataLoader(dataset, self.batch_size,
                                       shuffle=train)
```

```
@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)
```

The new data loader behaves just like the previous one, except that it is more efficient and has some added functionality.

```
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])
```

For instance, the data loader provided by the framework API supports the built-in `__len__` method, so we can query its length, i.e., the number of batches.

```
len(data.train_dataloader())
```

```
32
```

3.3.4 Summary

Data loaders are a convenient way of abstracting out the process of loading and manipulating data. This way the same machine learning *algorithm* is capable of processing many different types and sources of data without the need for modification. One of the nice things about data loaders is that they can be composed. For instance, we might be loading images and then have a postprocessing filter that crops them or modifies them in other ways. As such, data loaders can be used to describe an entire data processing pipeline.

As for the model itself, the two-dimensional linear model is about the simplest we might encounter. It lets us test out the accuracy of regression models without worrying about having insufficient amounts of data or an underdetermined system of equations. We will put this to good use in the next section.

3.3.5 Exercises

1. What will happen if the number of examples cannot be divided by the batch size. How would you change this behavior by specifying a different argument by using the framework's API?
2. Suppose that we want to generate a huge dataset, where both the size of the parameter vector `w` and the number of examples `num_examples` are large.
   1. What happens if we cannot hold all data in memory?
   2. How would you shuffle the data if it is held on disk? Your task is to design an *efficient* algorithm that does not require too many random reads or writes. Hint: pseudorandom permutation generators[75] allow you to design a reshuffle without the need to store the permutation table explicitly (Naor and Reingold, 1999).
3. Implement a data generator that produces new data on the fly, every time the iterator is called.
4. How would you design a random data generator that generates *the same* data each time it is called?

Discussions[76].

[75] 

[76] 

3.4 Linear Regression Implementation from Scratch

We are now ready to work through a fully functioning implementation of linear regression. In this section, we will implement the entire method from scratch, including (i) the model; (ii) the loss function; (iii) a minibatch stochastic gradient descent optimizer; and (iv) the training function that stitches all of these pieces together. Finally, we will run our synthetic data generator from Section 3.3 and apply our model on the resulting dataset. While modern deep learning frameworks can automate nearly all of this work, implementing things from scratch is the only way to make sure that you really know what you are doing. Moreover, when it is time to customize models, defining our own layers or loss functions, understanding how things work under the hood will prove handy. In this section, we will rely only on tensors and automatic differentiation. Later, we will introduce a more concise implementation, taking advantage of the bells and whistles of deep learning frameworks while retaining the structure of what follows below.

```
%matplotlib inline
import torch
from d2l import torch as d2l
```

3.4.1 Defining the Model

Before we can begin optimizing our model's parameters by minibatch SGD, we need to have some parameters in the first place. In the following we initialize weights by drawing random numbers from a normal distribution with mean 0 and a standard deviation of 0.01. The magic number 0.01 often works well in practice, but you can specify a different value through the argument `sigma`. Moreover we set the bias to 0. Note that for object-oriented design we add the code to the `__init__` method of a subclass of `d2l.Module` (introduced in Section 3.2.2).

```
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()
        self.w = torch.normal(0, sigma, (num_inputs, 1), requires_grad=True)
        self.b = torch.zeros(1, requires_grad=True)
```

Next we must define our model, relating its input and parameters to its output. Using the same notation as (3.1.4) for our linear model we simply take the matrix–vector product of the input features $\mathbf{X}$ and the model weights $\mathbf{w}$, and add the offset b to each example. The product $\mathbf{X}\mathbf{w}$ is a vector and b is a scalar. Because of the broadcasting mechanism (see Section 2.1.4), when we add a vector and a scalar, the scalar is added to each component of the vector. The resulting `forward` method is registered in the `LinearRegressionScratch` class via `add_to_class` (introduced in Section 3.2.1).

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

3.4.2 Defining the Loss Function

Since updating our model requires taking the gradient of our loss function, we ought to define the loss function first. Here we use the squared loss function in (3.1.5). In the implementation, we need to transform the true value `y` into the predicted value's shape `y_hat`. The result returned by the following method will also have the same shape as `y_hat`. We also return the averaged loss value among all examples in the minibatch.

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

3.4.3 Defining the Optimization Algorithm

As discussed in Section 3.1, linear regression has a closed-form solution. However, our goal here is to illustrate how to train more general neural networks, and that requires that we teach you how to use minibatch SGD. Hence we will take this opportunity to introduce your first working example of SGD. At each step, using a minibatch randomly drawn from our dataset, we estimate the gradient of the loss with respect to the parameters. Next, we update the parameters in the direction that may reduce the loss.

The following code applies the update, given a set of parameters, a learning rate `lr`. Since our loss is computed as an average over the minibatch, we do not need to adjust the learning rate against the batch size. In later chapters we will investigate how learning rates should be adjusted for very large minibatches as they arise in distributed large-scale learning. For now, we can ignore this dependency.

We define our `SGD` class, a subclass of `d2l.HyperParameters` (introduced in Section 3.2.1), to have a similar API as the built-in SGD optimizer. We update the parameters in the `step` method. The `zero_grad` method sets all gradients to 0, which must be run before a backpropagation step.

```
class SGD(d2l.HyperParameters):  #@save
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()

    def step(self):
        for param in self.params:
            param -= self.lr * param.grad

    def zero_grad(self):
```

(continues on next page)

(continued from previous page)

```
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

We next define the `configure_optimizers` method, which returns an instance of the `SGD` class.

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

3.4.4 Training

Now that we have all of the parts in place (parameters, loss function, model, and optimizer), we are ready to implement the main training loop. It is crucial that you understand this code fully since you will employ similar training loops for every other deep learning model covered in this book. In each *epoch*, we iterate through the entire training dataset, passing once through every example (assuming that the number of examples is divisible by the batch size). In each *iteration*, we grab a minibatch of training examples, and compute its loss through the model's `training_step` method. Then we compute the gradients with respect to each parameter. Finally, we will call the optimization algorithm to update the model parameters. In summary, we will execute the following loop:

- Initialize parameters $(\mathbf{w}, b)$
- Repeat until done
  - Compute gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$
  - Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

Recall that the synthetic regression dataset that we generated in Section 3.3 does not provide a validation dataset. In most cases, however, we will want a validation dataset to measure our model quality. Here we pass the validation dataloader once in each epoch to measure the model performance. Following our object-oriented design, the `prepare_batch` and `fit_epoch` methods are registered in the `d2l.Trainer` class (introduced in Section 3.2.4).

```
@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))
```

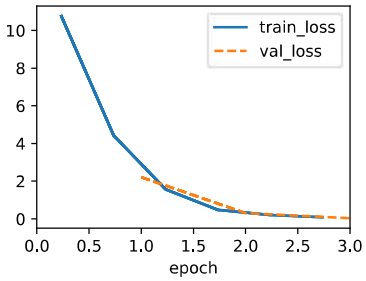
(continues on next page)

(continued from previous page)

```
        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
```

We are almost ready to train the model, but first we need some training data. Here we use the `SyntheticRegressionData` class and pass in some ground truth parameters. Then we train our model with the learning rate `lr=0.03` and set `max_epochs=3`. Note that in general, both the number of epochs and the learning rate are hyperparameters. In general, setting hyperparameters is tricky and we will usually want to use a three-way split, one set for training, a second for hyperparameter selection, and the third reserved for the final evaluation. We elide these details for now but will revise them later.

```
model = LinearRegressionScratch(2, lr=0.03)
data = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
trainer = d2l.Trainer(max_epochs=3)
trainer.fit(model, data)
```



Because we synthesized the dataset ourselves, we know precisely what the true parameters are. Thus, we can evaluate our success in training by comparing the true parameters with those that we learned through our training loop. Indeed they turn out to be very close to each other.

```
with torch.no_grad():
    print(f'error in estimating w: {data.w - model.w.reshape(data.w.shape)}')
    print(f'error in estimating b: {data.b - model.b}')
```

```
error in estimating w: tensor([ 0.1408, -0.1493])
error in estimating b: tensor([0.2130])
```

We should not take the ability to exactly recover the ground truth parameters for granted. In general, for deep models unique solutions for the parameters do not exist, and even for linear models, exactly recovering the parameters is only possible when no feature is linearly dependent on the others. However, in machine learning, we are often less concerned with recovering true underlying parameters, but rather with parameters that lead to highly accurate prediction (Vapnik, 1992). Fortunately, even on difficult optimization problems, stochastic gradient descent can often find remarkably good solutions, owing partly to the fact that, for deep networks, there exist many configurations of the parameters that lead to highly accurate prediction.

3.4.5 Summary

In this section, we took a significant step towards designing deep learning systems by implementing a fully functional neural network model and training loop. In this process, we built a data loader, a model, a loss function, an optimization procedure, and a visualization and monitoring tool. We did this by composing a Python object that contains all relevant components for training a model. While this is not yet a professional-grade implementation it is perfectly functional and code like this could already help you to solve small problems quickly. In the coming sections, we will see how to do this both *more concisely* (avoiding boilerplate code) and *more efficiently* (using our GPUs to their full potential).

3.4.6 Exercises

1. What would happen if we were to initialize the weights to zero. Would the algorithm still work? What if we initialized the parameters with variance 1000 rather than 0.01?
2. Assume that you are Georg Simon Ohm[77] trying to come up with a model for resistance that relates voltage and current. Can you use automatic differentiation to learn the parameters of your model?
3. Can you use Planck's Law[78] to determine the temperature of an object using spectral energy density? For reference, the spectral density B of radiation emanating from a black body is $B(\lambda, T) = \frac{2hc^2}{\lambda^5} \cdot \left(\exp \frac{hc}{\lambda kT} - 1\right)^{-1}$. Here λ is the wavelength, T is the temperature, c is the speed of light, h is Planck's constant, and k is the Boltzmann constant. You measure the energy for different wavelengths λ and you now need to fit the spectral density curve to Planck's law.
4. What are the problems you might encounter if you wanted to compute the second derivatives of the loss? How would you fix them?
5. Why is the `reshape` method needed in the `loss` function?
6. Experiment using different learning rates to find out how quickly the loss function value drops. Can you reduce the error by increasing the number of epochs of training?

[77] 

[78] 

7. If the number of examples cannot be divided by the batch size, what happens to `data_iter` at the end of an epoch?

8. Try implementing a different loss function, such as the absolute value loss `(y_hat - d2l.reshape(y, y_hat.shape)).abs().sum()`.

    1. Check what happens for regular data.

    2. Check whether there is a difference in behavior if you actively perturb some entries, such as $y_5 = 10000$, of $\mathbf{y}$.

    3. Can you think of a cheap solution for combining the best aspects of squared loss and absolute value loss? Hint: how can you avoid really large gradient values?

9. Why do we need to reshuffle the dataset? Can you design a case where a maliciously constructed dataset would break the optimization algorithm otherwise?

[79]  Discussions[79].

3.5 Concise Implementation of Linear Regression

Deep learning has witnessed a sort of Cambrian explosion over the past decade. The sheer number of techniques, applications and algorithms by far surpasses the progress of previous decades. This is due to a fortuitous combination of multiple factors, one of which is the powerful free tools offered by a number of open-source deep learning frameworks. Theano (Bergstra *et al.*, 2010), DistBelief (Dean *et al.*, 2012), and Caffe (Jia *et al.*, 2014) arguably represent the first generation of such models that found widespread adoption. In contrast to earlier (seminal) works like SN2 (Simulateur Neuristique) (Bottou and Le Cun, 1988), which provided a Lisp-like programming experience, modern frameworks offer automatic differentiation and the convenience of Python. These frameworks allow us to automate and modularize the repetitive work of implementing gradient-based learning algorithms.

In Section 3.4, we relied only on (i) tensors for data storage and linear algebra; and (ii) automatic differentiation for calculating gradients. In practice, because data iterators, loss functions, optimizers, and neural network layers are so common, modern libraries implement these components for us as well. In this section, we will show you how to implement the linear regression model from Section 3.4 concisely by using high-level APIs of deep learning frameworks.

```
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

3.5.1 Defining the Model

When we implemented linear regression from scratch in Section 3.4, we defined our model parameters explicitly and coded up the calculations to produce output using basic linear algebra operations. You *should* know how to do this. But once your models get more complex, and once you have to do this nearly every day, you will be glad of the assistance. The situation is similar to coding up your own blog from scratch. Doing it once or twice is rewarding and instructive, but you would be a lousy web developer if you spent a month reinventing the wheel.

For standard operations, we can use a framework's predefined layers, which allow us to focus on the layers used to construct the model rather than worrying about their implementation. Recall the architecture of a single-layer network as described in Fig. 3.1.2. The layer is called *fully connected*, since each of its inputs is connected to each of its outputs by means of a matrix–vector multiplication.

In PyTorch, the fully connected layer is defined in `Linear` and `LazyLinear` classes (available since version 1.8.0). The latter allows users to specify *merely* the output dimension, while the former additionally asks for how many inputs go into this layer. Specifying input shapes is inconvenient and may require nontrivial calculations (such as in convolutional layers). Thus, for simplicity, we will use such "lazy" layers whenever we can.

```
class LinearRegression(d2l.Module):  #@save
    """The linear regression model implemented with high-level APIs."""
    def __init__(self, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.LazyLinear(1)
        self.net.weight.data.normal_(0, 0.01)
        self.net.bias.data.fill_(0)
```

In the `forward` method we just invoke the built-in `__call__` method of the predefined layers to compute the outputs.

```
@d2l.add_to_class(LinearRegression)  #@save
def forward(self, X):
    return self.net(X)
```

3.5.2 Defining the Loss Function

The `MSELoss` class computes the mean squared error (without the $1/2$ factor in $(3.1.5)$). By default, `MSELoss` returns the average loss over examples. It is faster (and easier to use) than implementing our own.

```
@d2l.add_to_class(LinearRegression)  #@save
def loss(self, y_hat, y):
    fn = nn.MSELoss()
    return fn(y_hat, y)
```

3.5.3 Defining the Optimization Algorithm

Minibatch SGD is a standard tool for optimizing neural networks and thus PyTorch supports it alongside a number of variations on this algorithm in the `optim` module. When we instantiate an `SGD` instance, we specify the parameters to optimize over, obtainable from our model via `self.parameters()`, and the learning rate (`self.lr`) required by our optimization algorithm.

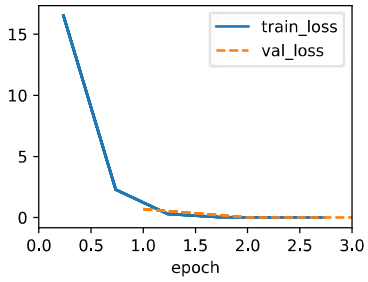
```
@d2l.add_to_class(LinearRegression)  #@save
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), self.lr)
```

3.5.4 Training

You might have noticed that expressing our model through high-level APIs of a deep learning framework requires fewer lines of code. We did not have to allocate parameters individually, define our loss function, or implement minibatch SGD. Once we start working with much more complex models, the advantages of the high-level API will grow considerably.

Now that we have all the basic pieces in place, the training loop itself is the same as the one we implemented from scratch. So we just call the `fit` method (introduced in Section 3.2.4), which relies on the implementation of the `fit_epoch` method in Section 3.4, to train our model.

```
model = LinearRegression(lr=0.03)
data = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
trainer = d2l.Trainer(max_epochs=3)
trainer.fit(model, data)
```



Below, we compare the model parameters learned by training on finite data and the actual parameters that generated our dataset. To access parameters, we access the weights and bias of the layer that we need. As in our implementation from scratch, note that our estimated parameters are close to their true counterparts.

```
@d2l.add_to_class(LinearRegression)  #@save
```

(continues on next page)

(continued from previous page)

```
def get_w_b(self):
    return (self.net.weight.data, self.net.bias.data)
w, b = model.get_w_b()
```

```
print(f'error in estimating w: {data.w - w.reshape(data.w.shape)}')
print(f'error in estimating b: {data.b - b}')
```

```
error in estimating w: tensor([ 0.0094, -0.0030])
error in estimating b: tensor([0.0137])
```

3.5.5 Summary

This section contains the first implementation of a deep network (in this book) to tap into the conveniences afforded by modern deep learning frameworks, such as MXNet (Chen *et al.*, 2015), JAX (Frostig *et al.*, 2018), PyTorch (Paszke *et al.*, 2019), and Tensorflow (Abadi *et al.*, 2016). We used framework defaults for loading data, defining a layer, a loss function, an optimizer and a training loop. Whenever the framework provides all necessary features, it is generally a good idea to use them, since the library implementations of these components tend to be heavily optimized for performance and properly tested for reliability. At the same time, try not to forget that these modules *can* be implemented directly. This is especially important for aspiring researchers who wish to live on the leading edge of model development, where you will be inventing new components that cannot possibly exist in any current library.

In PyTorch, the `data` module provides tools for data processing, the `nn` module defines a large number of neural network layers and common loss functions. We can initialize the parameters by replacing their values with methods ending with `_`. Note that we need to specify the input dimensions of the network. While this is trivial for now, it can have significant knock-on effects when we want to design complex networks with many layers. Careful considerations of how to parametrize these networks is needed to allow portability.

3.5.6 Exercises

1. How would you need to change the learning rate if you replace the aggregate loss over the minibatch with an average over the loss on the minibatch?
2. Review the framework documentation to see which loss functions are provided. In particular, replace the squared loss with Huber's robust loss function. That is, use the loss function

$$l(y, y') = \begin{cases} |y - y'| - \frac{\sigma}{2} & \text{if } |y - y'| > \sigma \\ \frac{1}{2\sigma}(y - y')^2 & \text{otherwise} \end{cases} \tag{3.5.1}$$

3. How do you access the gradient of the weights of the model?

4. What is the effect on the solution if you change the learning rate and the number of epochs? Does it keep on improving?
5. How does the solution change as you vary the amount of data generated?
    1. Plot the estimation error for $\hat{\mathbf{w}} - \mathbf{w}$ and $\hat{b} - b$ as a function of the amount of data. Hint: increase the amount of data logarithmically rather than linearly, i.e., 5, 10, 20, 50, …, 10,000 rather than 1000, 2000, …, 10,000.
    2. Why is the suggestion in the hint appropriate?

Discussions[80].

3.6 Generalization

Consider two college students diligently preparing for their final exam. Commonly, this preparation will consist of practicing and testing their abilities by taking exams administered in previous years. Nonetheless, doing well on past exams is no guarantee that they will excel when it matters. For instance, imagine a student, Extraordinary Ellie, whose preparation consisted entirely of memorizing the answers to previous years' exam questions. Even if Ellie were endowed with an extraordinary memory, and thus could perfectly recall the answer to any *previously seen* question, she might nevertheless freeze when faced with a new (*previously unseen*) question. By comparison, imagine another student, Inductive Irene, with comparably poor memorization skills, but a knack for picking up patterns. Note that if the exam truly consisted of recycled questions from a previous year, Ellie would handily outperform Irene. Even if Irene's inferred patterns yielded 90% accurate predictions, they could never compete with Ellie's 100% recall. However, even if the exam consisted entirely of fresh questions, Irene might maintain her 90% average.

As machine learning scientists, our goal is to discover *patterns*. But how can we be sure that we have truly discovered a *general* pattern and not simply memorized our data? Most of the time, our predictions are only useful if our model discovers such a pattern. We do not want to predict yesterday's stock prices, but tomorrow's. We do not need to recognize already diagnosed diseases for previously seen patients, but rather previously undiagnosed ailments in previously unseen patients. This problem—how to discover patterns that *generalize*—is the fundamental problem of machine learning, and arguably of all of statistics. We might cast this problem as just one slice of a far grander question that engulfs all of science: when are we ever justified in making the leap from particular observations to more general statements?

In real life, we must fit our models using a finite collection of data. The typical scales of that data vary wildly across domains. For many important medical problems, we can only access a few thousand data points. When studying rare diseases, we might be lucky to access hundreds. By contrast, the largest public datasets consisting of labeled photographs, e.g., ImageNet (Deng *et al.*, 2009), contain millions of images. And some unlabeled image

collections such as the Flickr YFC100M dataset can be even larger, containing over 100 million images (Thomee *et al.*, 2016). However, even at this extreme scale, the number of available data points remains infinitesimally small compared to the space of all possible images at a megapixel resolution. Whenever we work with finite samples, we must keep in mind the risk that we might fit our training data, only to discover that we failed to discover a generalizable pattern.

The phenomenon of fitting closer to our training data than to the underlying distribution is called *overfitting*, and techniques for combatting overfitting are often called *regularization* methods. While it is no substitute for a proper introduction to statistical learning theory (see Boucheron *et al.* (2005), Vapnik (1998)), we will give you just enough intuition to get going. We will revisit generalization in many chapters throughout the book, exploring both what is known about the principles underlying generalization in various models, and also heuristic techniques that have been found (empirically) to yield improved generalization on tasks of practical interest.

3.6.1 Training Error and Generalization Error

In the standard supervised learning setting, we assume that the training data and the test data are drawn *independently* from *identical* distributions. This is commonly called the *IID assumption*. While this assumption is strong, it is worth noting that, absent any such assumption, we would be dead in the water. Why should we believe that training data sampled from distribution $P(X, Y)$ should tell us how to make predictions on test data generated by a *different distribution* $Q(X, Y)$? Making such leaps turns out to require strong assumptions about how P and Q are related. Later on we will discuss some assumptions that allow for shifts in distribution but first we need to understand the IID case, where $P(\cdot) = Q(\cdot)$.

To begin with, we need to differentiate between the *training error* R_{emp}, which is a *statistic* calculated on the training dataset, and the *generalization error* R, which is an *expectation* taken with respect to the underlying distribution. You can think of the generalization error as what you would see if you applied your model to an infinite stream of additional data examples drawn from the same underlying data distribution. Formally the training error is expressed as a *sum* (with the same notation as Section 3.1):

$$R_{\text{emp}}[\mathbf{X}, \mathbf{y}, f] = \frac{1}{n} \sum_{i=1}^{n} l(\mathbf{x}^{(i)}, y^{(i)}, f(\mathbf{x}^{(i)})), \tag{3.6.1}$$

while the generalization error is expressed as an integral:

$$R[p, f] = E_{(\mathbf{x},y)\sim P}[l(\mathbf{x}, y, f(\mathbf{x}))] = \int \int l(\mathbf{x}, y, f(\mathbf{x}))p(\mathbf{x}, y)\, d\mathbf{x}dy. \tag{3.6.2}$$

Problematically, we can never calculate the generalization error R exactly. Nobody ever tells us the precise form of the density function $p(\mathbf{x}, y)$. Moreover, we cannot sample an infinite stream of data points. Thus, in practice, we must *estimate* the generalization error by applying our model to an independent test set constituted of a random selection of examples $\mathbf{X}'$ and labels $\mathbf{y}'$ that were withheld from our training set. This consists of

applying the same formula that was used for calculating the empirical training error but to a test set $\mathbf{X}', \mathbf{y}'$.

Crucially, when we evaluate our classifier on the test set, we are working with a *fixed* classifier (it does not depend on the sample of the test set), and thus estimating its error is simply the problem of mean estimation. However the same cannot be said for the training set. Note that the model we wind up with depends explicitly on the selection of the training set and thus the training error will in general be a biased estimate of the true error on the underlying population. The central question of generalization is then when should we expect our training error to be close to the population error (and thus the generalization error).

Model Complexity

In classical theory, when we have simple models and abundant data, the training and generalization errors tend to be close. However, when we work with more complex models and/or fewer examples, we expect the training error to go down but the generalization gap to grow. This should not be surprising. Imagine a model class so expressive that for any dataset of n examples, we can find a set of parameters that can perfectly fit arbitrary labels, even if randomly assigned. In this case, even if we fit our training data perfectly, how can we conclude anything about the generalization error? For all we know, our generalization error might be no better than random guessing.

In general, absent any restriction on our model class, we cannot conclude, based on fitting the training data alone, that our model has discovered any generalizable pattern (Vapnik *et al.*, 1994). On the other hand, if our model class was not capable of fitting arbitrary labels, then it must have discovered a pattern. Learning-theoretic ideas about model complexity derived some inspiration from the ideas of Karl Popper, an influential philosopher of science, who formalized the criterion of falsifiability. According to Popper, a theory that can explain any and all observations is not a scientific theory at all! After all, what has it told us about the world if it has not ruled out any possibility? In short, what we want is a hypothesis that *could not* explain any observations we might conceivably make and yet nevertheless happens to be compatible with those observations that we *in fact* make.

Now what precisely constitutes an appropriate notion of model complexity is a complex matter. Often, models with more parameters are able to fit a greater number of arbitrarily assigned labels. However, this is not necessarily true. For instance, kernel methods operate in spaces with infinite numbers of parameters, yet their complexity is controlled by other means (Schölkopf and Smola, 2002). One notion of complexity that often proves useful is the range of values that the parameters can take. Here, a model whose parameters are permitted to take arbitrary values would be more complex. We will revisit this idea in the next section, when we introduce *weight decay*, your first practical regularization technique. Notably, it can be difficult to compare complexity among members of substantially different model classes (say, decision trees vs. neural networks).

At this point, we must stress another important point that we will revisit when introducing deep neural networks. When a model is capable of fitting arbitrary labels, low training error does not necessarily imply low generalization error. *However, it does not necessarily*

imply high generalization error either! All we can say with confidence is that low training error alone is not enough to certify low generalization error. Deep neural networks turn out to be just such models: while they generalize well in practice, they are too powerful to allow us to conclude much on the basis of training error alone. In these cases we must rely more heavily on our holdout data to certify generalization after the fact. Error on the holdout data, i.e., validation set, is called the *validation error*.

3.6.2 Underfitting or Overfitting?

When we compare the training and validation errors, we want to be mindful of two common situations. First, we want to watch out for cases when our training error and validation error are both substantial but there is a little gap between them. If the model is unable to reduce the training error, that could mean that our model is too simple (i.e., insufficiently expressive) to capture the pattern that we are trying to model. Moreover, since the *generalization gap* ($R_{\text{emp}} - R$) between our training and generalization errors is small, we have reason to believe that we could get away with a more complex model. This phenomenon is known as *underfitting*.

On the other hand, as we discussed above, we want to watch out for the cases when our training error is significantly lower than our validation error, indicating severe *overfitting*. Note that overfitting is not always a bad thing. In deep learning especially, the best predictive models often perform far better on training data than on holdout data. Ultimately, we usually care about driving the generalization error lower, and only care about the gap insofar as it becomes an obstacle to that end. Note that if the training error is zero, then the generalization gap is precisely equal to the generalization error and we can make progress only by reducing the gap.

Polynomial Curve Fitting

To illustrate some classical intuition about overfitting and model complexity, consider the following: given training data consisting of a single feature x and a corresponding real-valued label y, we try to find the polynomial of degree d

$$\hat{y} = \sum_{i=0}^{d} x^i w_i \tag{3.6.3}$$

for estimating the label y. This is just a linear regression problem where our features are given by the powers of x, the model's weights are given by w_i, and the bias is given by w_0 since $x^0 = 1$ for all x. Since this is just a linear regression problem, we can use the squared error as our loss function.

A higher-order polynomial function is more complex than a lower-order polynomial function, since the higher-order polynomial has more parameters and the model function's selection range is wider. Fixing the training dataset, higher-order polynomial functions should always achieve lower (at worst, equal) training error relative to lower-degree polynomials. In fact, whenever each data example has a distinct value of x, a polynomial function with degree equal to the number of data examples can fit the training set perfectly. We compare

the relationship between polynomial degree (model complexity) and both underfitting and overfitting in Fig. 3.6.1.

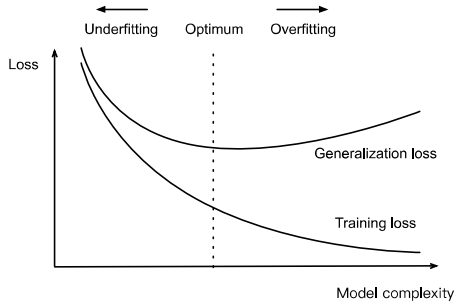


Fig. 3.6.1 Influence of model complexity on underfitting and overfitting.

Dataset Size

As the above bound already indicates, another big consideration to bear in mind is dataset size. Fixing our model, the fewer samples we have in the training dataset, the more likely (and more severely) we are to encounter overfitting. As we increase the amount of training data, the generalization error typically decreases. Moreover, in general, more data never hurts. For a fixed task and data distribution, model complexity should not increase more rapidly than the amount of data. Given more data, we might attempt to fit a more complex model. Absent sufficient data, simpler models may be more difficult to beat. For many tasks, deep learning only outperforms linear models when many thousands of training examples are available. In part, the current success of deep learning owes considerably to the abundance of massive datasets arising from Internet companies, cheap storage, connected devices, and the broad digitization of the economy.

3.6.3 Model Selection

Typically, we select our final model only after evaluating multiple models that differ in various ways (different architectures, training objectives, selected features, data preprocessing, learning rates, etc.). Choosing among many models is aptly called *model selection*.

In principle, we should not touch our test set until after we have chosen all our hyperparameters. Were we to use the test data in the model selection process, there is a risk that we might overfit the test data. Then we would be in serious trouble. If we overfit our training data, there is always the evaluation on test data to keep us honest. But if we overfit the test data, how would we ever know? See Ong *et al.* (2005) for an example of how this can lead to absurd results even for models where the complexity can be tightly controlled.

Thus, we should never rely on the test data for model selection. And yet we cannot rely solely on the training data for model selection either because we cannot estimate the generalization error on the very data that we use to train the model.

In practical applications, the picture gets muddier. While ideally we would only touch the

test data once, to assess the very best model or to compare a small number of models with each other, real-world test data is seldom discarded after just one use. We can seldom afford a new test set for each round of experiments. In fact, recycling benchmark data for decades can have a significant impact on the development of algorithms, e.g., for image classification[81] and optical character recognition[82].

[81] The common practice for addressing the problem of *training on the test set* is to split our data three ways, incorporating a *validation set* in addition to the training and test datasets. The result is a murky business where the boundaries between validation and test data are worryingly ambiguous. Unless explicitly stated otherwise, in the experiments in this book we are really working with what should rightly be called training data and validation data, with no true test sets. Therefore, the accuracy reported in each experiment of the book is really the validation accuracy and not a true test set accuracy.

[82]

Cross-Validation

When training data is scarce, we might not even be able to afford to hold out enough data to constitute a proper validation set. One popular solution to this problem is to employ *K-fold cross-validation*. Here, the original training data is split into K non-overlapping subsets. Then model training and validation are executed K times, each time training on $K - 1$ subsets and validating on a different subset (the one not used for training in that round). Finally, the training and validation errors are estimated by averaging over the results from the K experiments.

3.6.4 Summary

This section explored some of the underpinnings of generalization in machine learning. Some of these ideas become complicated and counterintuitive when we get to deeper models; here, models are capable of overfitting data badly, and the relevant notions of complexity can be both implicit and counterintuitive (e.g., larger architectures with more parameters generalizing better). We leave you with a few rules of thumb:

1. Use validation sets (or *K-fold cross-validation*) for model selection;
2. More complex models often require more data;
3. Relevant notions of complexity include both the number of parameters and the range of values that they are allowed to take;
4. Keeping all else equal, more data almost always leads to better generalization;
5. This entire talk of generalization is all predicated on the IID assumption. If we relax this assumption, allowing for distributions to shift between the train and testing periods, then we cannot say anything about generalization absent a further (perhaps milder) assumption.

3.6.5 Exercises

1. When can you solve the problem of polynomial regression exactly?

2. Give at least five examples where dependent random variables make treating the problem as IID data inadvisable.

3. Can you ever expect to see zero training error? Under which circumstances would you see zero generalization error?

4. Why is K-fold cross-validation very expensive to compute?

5. Why is the K-fold cross-validation error estimate biased?

6. The VC dimension is defined as the maximum number of points that can be classified with arbitrary labels $\{\pm 1\}$ by a function of a class of functions. Why might this not be a good idea for measuring how complex the class of functions is? Hint: consider the magnitude of the functions.

7. Your manager gives you a difficult dataset on which your current algorithm does not perform so well. How would you justify to him that you need more data? Hint: you cannot increase the data but you can decrease it.

[83]  Discussions[83].

3.7 Weight Decay

Now that we have characterized the problem of overfitting, we can introduce our first *regularization* technique. Recall that we can always mitigate overfitting by collecting more training data. However, that can be costly, time consuming, or entirely out of our control, making it impossible in the short run. For now, we can assume that we already have as much high-quality data as our resources permit and focus the tools at our disposal when the dataset is taken as a given.

Recall that in our polynomial regression example (Section 3.6.2) we could limit our model's capacity by tweaking the degree of the fitted polynomial. Indeed, limiting the number of features is a popular technique for mitigating overfitting. However, simply tossing aside features can be too blunt an instrument. Sticking with the polynomial regression example, consider what might happen with high-dimensional input. The natural extensions of polynomials to multivariate data are called *monomials*, which are simply products of powers of variables. The degree of a monomial is the sum of the powers. For example, $x_1^2 x_2$, and $x_3 x_5^2$ are both monomials of degree 3.

Note that the number of terms with degree d blows up rapidly as d grows larger. Given k variables, the number of monomials of degree d is $\binom{k-1+d}{k-1}$. Even small changes in degree, say from 2 to 3, dramatically increase the complexity of our model. Thus we often need a more fine-grained tool for adjusting function complexity.

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

3.7.1 Norms and Weight Decay

Rather than directly manipulating the number of parameters, *weight decay*, operates by restricting the values that the parameters can take. More commonly called ℓ_2 regularization outside of deep learning circles when optimized by minibatch stochastic gradient descent, weight decay might be the most widely used technique for regularizing parametric machine learning models. The technique is motivated by the basic intuition that among all functions f, the function $f = 0$ (assigning the value 0 to all inputs) is in some sense the *simplest*, and that we can measure the complexity of a function by the distance of its parameters from zero. But how precisely should we measure the distance between a function and zero? There is no single right answer. In fact, entire branches of mathematics, including parts of functional analysis and the theory of Banach spaces, are devoted to addressing such issues.

One simple interpretation might be to measure the complexity of a linear function $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ by some norm of its weight vector, e.g., $\|\mathbf{w}\|^2$. Recall that we introduced the ℓ_2 norm and ℓ_1 norm, which are special cases of the more general ℓ_p norm, in Section 2.3.11. The most common method for ensuring a small weight vector is to add its norm as a penalty term to the problem of minimizing the loss. Thus we replace our original objective, *minimizing the prediction loss on the training labels*, with new objective, *minimizing the sum of the prediction loss and the penalty term*. Now, if our weight vector grows too large, our learning algorithm might focus on minimizing the weight norm $\|\mathbf{w}\|^2$ rather than minimizing the training error. That is exactly what we want. To illustrate things in code, we revive our previous example from Section 3.1 for linear regression. There, our loss was given by

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{3.7.1}$$

Recall that $\mathbf{x}^{(i)}$ are the features, $y^{(i)}$ is the label for any data example i, and $(\mathbf{w}, b)$ are the weight and bias parameters, respectively. To penalize the size of the weight vector, we must somehow add $\|\mathbf{w}\|^2$ to the loss function, but how should the model trade off the standard loss for this new additive penalty? In practice, we characterize this trade-off via the *regularization constant* λ, a nonnegative hyperparameter that we fit using validation data:

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2. \tag{3.7.2}$$

For $\lambda = 0$, we recover our original loss function. For $\lambda > 0$, we restrict the size of $\|\mathbf{w}\|$. We divide by 2 by convention: when we take the derivative of a quadratic function, the 2 and $1/2$ cancel out, ensuring that the expression for the update looks nice and simple. The astute reader might wonder why we work with the squared norm and not the standard

norm (i.e., the Euclidean distance). We do this for computational convenience. By squaring the ℓ_2 norm, we remove the square root, leaving the sum of squares of each component of the weight vector. This makes the derivative of the penalty easy to compute: the sum of derivatives equals the derivative of the sum.

Moreover, you might ask why we work with the ℓ_2 norm in the first place and not, say, the ℓ_1 norm. In fact, other choices are valid and popular throughout statistics. While ℓ_2-regularized linear models constitute the classic *ridge regression* algorithm, ℓ_1-regularized linear regression is a similarly fundamental method in statistics, popularly known as *lasso regression*. One reason to work with the ℓ_2 norm is that it places an outsize penalty on large components of the weight vector. This biases our learning algorithm towards models that distribute weight evenly across a larger number of features. In practice, this might make them more robust to measurement error in a single variable. By contrast, ℓ_1 penalties lead to models that concentrate weights on a small set of features by clearing the other weights to zero. This gives us an effective method for *feature selection*, which may be desirable for other reasons. For example, if our model only relies on a few features, then we may not need to collect, store, or transmit data for the other (dropped) features.

Using the same notation in $(3.1.11)$, minibatch stochastic gradient descent updates for ℓ_2-regularized regression as follows:

$$\mathbf{w} \leftarrow (1 - \eta\lambda)\,\mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \mathbf{x}^{(i)}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right). \tag{3.7.3}$$

As before, we update $\mathbf{w}$ based on the amount by which our estimate differs from the observation. However, we also shrink the size of $\mathbf{w}$ towards zero. That is why the method is sometimes called "weight decay": given the penalty term alone, our optimization algorithm *decays* the weight at each step of training. In contrast to feature selection, weight decay offers us a mechanism for continuously adjusting the complexity of a function. Smaller values of λ correspond to less constrained $\mathbf{w}$, whereas larger values of λ constrain $\mathbf{w}$ more considerably. Whether we include a corresponding bias penalty b^2 can vary across implementations, and may vary across layers of a neural network. Often, we do not regularize the bias term. Besides, although ℓ_2 regularization may not be equivalent to weight decay for other optimization algorithms, the idea of regularization through shrinking the size of weights still holds true.

3.7.2 High-Dimensional Linear Regression

We can illustrate the benefits of weight decay through a simple synthetic example.

First, we generate some data as before:

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01^2). \tag{3.7.4}$$

In this synthetic dataset, our label is given by an underlying linear function of our inputs, corrupted by Gaussian noise with zero mean and standard deviation 0.01. For illustrative purposes, we can make the effects of overfitting pronounced, by increasing the dimen-

sionality of our problem to $d = 200$ and working with a small training set with only 20 examples.

```
class Data(d2l.DataModule):
    def __init__(self, num_train, num_val, num_inputs, batch_size):
        self.save_hyperparameters()
        n = num_train + num_val
        self.X = torch.randn(n, num_inputs)
        noise = torch.randn(n, 1) * 0.01
        w, b = torch.ones((num_inputs, 1)) * 0.01, 0.05
        self.y = torch.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)
```

3.7.3 Implementation from Scratch

Now, let's try implementing weight decay from scratch. Since minibatch stochastic gradient descent is our optimizer, we just need to add the squared ℓ_2 penalty to the original loss function.

Defining ℓ_2 Norm Penalty

Perhaps the most convenient way of implementing this penalty is to square all terms in place and sum them.

```
def l2_penalty(w):
    return (w ** 2).sum() / 2
```

Defining the Model

In the final model, the linear regression and the squared loss have not changed since Section 3.4, so we will just define a subclass of `d2l.LinearRegressionScratch`. The only change here is that our loss now includes the penalty term.

```
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))
```

The following code fits our model on the training set with 20 examples and evaluates it on the validation set with 100 examples.

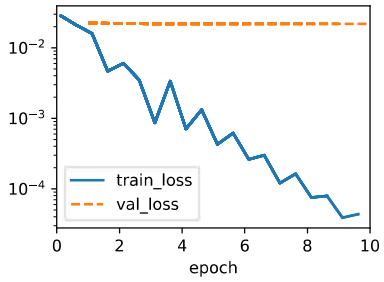
```
data = Data(num_train=20, num_val=100, num_inputs=200, batch_size=5)
trainer = d2l.Trainer(max_epochs=10)

def train_scratch(lambd):
    model = WeightDecayScratch(num_inputs=200, lambd=lambd, lr=0.01)
    model.board.yscale='log'
    trainer.fit(model, data)
    print('L2 norm of w:', float(l2_penalty(model.w)))
```

Training without Regularization

We now run this code with `lambd = 0`, disabling weight decay. Note that we overfit badly, decreasing the training error but not the validation error—a textbook case of overfitting.

```
train_scratch(0)
```

```
L2 norm of w: 0.009948714636266232
```



Using Weight Decay

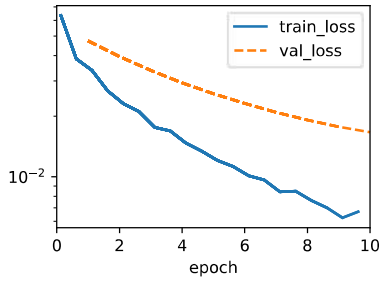
Below, we run with substantial weight decay. Note that the training error increases but the validation error decreases. This is precisely the effect we expect from regularization.

```
train_scratch(3)
```

```
L2 norm of w: 0.0017270983662456274
```

3.7.4 Concise Implementation

Because weight decay is ubiquitous in neural network optimization, the deep learning framework makes it especially convenient, integrating weight decay into the optimization



algorithm itself for easy use in combination with any loss function. Moreover, this integration serves a computational benefit, allowing implementation tricks to add weight decay to the algorithm, without any additional computational overhead. Since the weight decay portion of the update depends only on the current value of each parameter, the optimizer must touch each parameter once anyway.

Below, we specify the weight decay hyperparameter directly through `weight_decay` when instantiating our optimizer. By default, PyTorch decays both weights and biases simultaneously, but we can configure the optimizer to handle different parameters according to different policies. Here, we only set `weight_decay` for the weights (the `net.weight` parameters), hence the bias (the `net.bias` parameter) will not decay.

```
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)
```

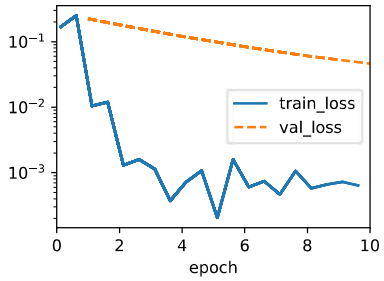
The plot looks similar to that when we implemented weight decay from scratch. However, this version runs faster and is easier to implement, benefits that will become more pronounced as you address larger problems and this work becomes more routine.

```
model = WeightDecay(wd=3, lr=0.01)
model.board.yscale='log'
trainer.fit(model, data)

print('L2 norm of w:', float(l2_penalty(model.get_w_b()[0])))
```

```
L2 norm of w: 0.013779522851109505
```

So far, we have touched upon one notion of what constitutes a simple linear function. However, even for simple nonlinear functions, the situation can be much more complex. To see this, the concept of reproducing kernel Hilbert space (RKHS)[84] allows one to apply tools

84



introduced for linear functions in a nonlinear context. Unfortunately, RKHS-based algorithms tend to scale poorly to large, high-dimensional data. In this book we will often adopt the common heuristic whereby weight decay is applied to all layers of a deep network.

3.7.5 Summary

Regularization is a common method for dealing with overfitting. Classical regularization techniques add a penalty term to the loss function (when training) to reduce the complexity of the learned model. One particular choice for keeping the model simple is using an ℓ_2 penalty. This leads to weight decay in the update steps of the minibatch stochastic gradient descent algorithm. In practice, the weight decay functionality is provided in optimizers from deep learning frameworks. Different sets of parameters can have different update behaviors within the same training loop.

3.7.6 Exercises

1. Experiment with the value of λ in the estimation problem in this section. Plot training and validation accuracy as a function of λ. What do you observe?
2. Use a validation set to find the optimal value of λ. Is it really the optimal value? Does this matter?
3. What would the update equations look like if instead of $\|\mathbf{w}\|^2$ we used $\sum_i |w_i|$ as our penalty of choice (ℓ_1 regularization)?
4. We know that $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$. Can you find a similar equation for matrices (see the Frobenius norm in Section 2.3.11)?
5. Review the relationship between training error and generalization error. In addition to weight decay, increased training, and the use of a model of suitable complexity, what other ways might help us deal with overfitting?
6. In Bayesian statistics we use the product of prior and likelihood to arrive at a posterior via $P(w \mid x) \propto P(x \mid w)P(w)$. How can you identify $P(w)$ with regularization?

Discussions[85].

85 

4 Linear Neural Networks for Classification

Now that you have worked through all of the mechanics you are ready to apply the skills you have learned to broader kinds of tasks. Even as we pivot towards classification, most of the plumbing remains the same: loading the data, passing it through the model, generating output, calculating the loss, taking gradients with respect to weights, and updating the model. However, the precise form of the targets, the parametrization of the output layer, and the choice of loss function will adapt to suit the *classification* setting.

4.1 Softmax Regression

In Section 3.1, we introduced linear regression, working through implementations from scratch in Section 3.4 and again using high-level APIs of a deep learning framework in Section 3.5 to do the heavy lifting.

Regression is the hammer we reach for when we want to answer *how much?* or *how many?* questions. If you want to predict the number of dollars (price) at which a house will be sold, or the number of wins a baseball team might have, or the number of days that a patient will remain hospitalized before being discharged, then you are probably looking for a regression model. However, even within regression models, there are important distinctions. For instance, the price of a house will never be negative and changes might often be *relative* to its baseline price. As such, it might be more effective to regress on the logarithm of the price. Likewise, the number of days a patient spends in hospital is a *discrete nonnegative* random variable. As such, least mean squares might not be an ideal approach either. This sort of time-to-event modeling comes with a host of other complications that are dealt with in a specialized subfield called *survival modeling*.

The point here is not to overwhelm you but just to let you know that there is a lot more to estimation than simply minimizing squared errors. And more broadly, there is a lot more to supervised learning than regression. In this section, we focus on *classification* problems where we put aside *how much?* questions and instead focus on *which category?* questions.

- Does this email belong in the spam folder or the inbox?
- Is this customer more likely to sign up or not to sign up for a subscription service?

- Does this image depict a donkey, a dog, a cat, or a rooster?
- Which movie is Aston most likely to watch next?
- Which section of the book are you going to read next?

Colloquially, machine learning practitioners overload the word *classification* to describe two subtly different problems: (i) those where we are interested only in hard assignments of examples to categories (classes); and (ii) those where we wish to make soft assignments, i.e., to assess the probability that each category applies. The distinction tends to get blurred, in part, because often, even when we only care about hard assignments, we still use models that make soft assignments.

Even more, there are cases where more than one label might be true. For instance, a news article might simultaneously cover the topics of entertainment, business, and space flight, but not the topics of medicine or sports. Thus, categorizing it into one of the above categories on their own would not be very useful. This problem is commonly known as multi-label classification[86]. See Tsoumakas and Katakis (2007) for an overview and Huang *et al.* (2015) for an effective algorithm when tagging images.

4.1.1 Classification

To get our feet wet, let's start with a simple image classification problem. Here, each input consists of a 2×2 grayscale image. We can represent each pixel value with a single scalar, giving us four features x_1, x_2, x_3, x_4. Further, let's assume that each image belongs to one among the categories "cat", "chicken", and "dog".

Next, we have to choose how to represent the labels. We have two obvious choices. Perhaps the most natural impulse would be to choose $y \in \{1, 2, 3\}$, where the integers represent $\{\text{dog}, \text{cat}, \text{chicken}\}$ respectively. This is a great way of *storing* such information on a computer. If the categories had some natural ordering among them, say if we were trying to predict $\{\text{baby}, \text{toddler}, \text{adolescent}, \text{young adult}, \text{adult}, \text{geriatric}\}$, then it might even make sense to cast this as an ordinal regression[87] problem and keep the labels in this format. See Moon *et al.* (2010) for an overview of different types of ranking loss functions and Beutel *et al.* (2014) for a Bayesian approach that addresses responses with more than one mode.

In general, classification problems do not come with natural orderings among the classes. Fortunately, statisticians long ago invented a simple way to represent categorical data: the *one-hot encoding*. A one-hot encoding is a vector with as many components as we have categories. The component corresponding to a particular instance's category is set to 1 and all other components are set to 0. In our case, a label y would be a three-dimensional vector, with $(1, 0, 0)$ corresponding to "cat", $(0, 1, 0)$ to "chicken", and $(0, 0, 1)$ to "dog":

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \tag{4.1.1}$$

Linear Model

In order to estimate the conditional probabilities associated with all the possible classes, we need a model with multiple outputs, one per class. To address classification with linear models, we will need as many affine functions as we have outputs. Strictly speaking, we only need one fewer, since the final category has to be the difference between 1 and the sum of the other categories, but for reasons of symmetry we use a slightly redundant parametrization. Each output corresponds to its own affine function. In our case, since we have 4 features and 3 possible output categories, we need 12 scalars to represent the weights (w with subscripts), and 3 scalars to represent the biases (b with subscripts). This yields:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned} \tag{4.1.2}$$

The corresponding neural network diagram is shown in Fig. 4.1.1. Just as in linear regression, we use a single-layer neural network. And since the calculation of each output, o_1, o_2, and o_3, depends on every input, x_1, x_2, x_3, and x_4, the output layer can also be described as a *fully connected layer*.

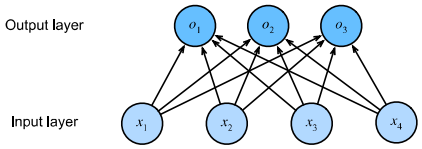


Fig. 4.1.1 Softmax regression is a single-layer neural network.

For a more concise notation we use vectors and matrices: $\mathbf{o} = \mathbf{W}\mathbf{x}+\mathbf{b}$ is much better suited for mathematics and code. Note that we have gathered all of our weights into a 3×4 matrix and all biases $\mathbf{b} \in \mathbb{R}^3$ in a vector.

The Softmax

Assuming a suitable loss function, we could try, directly, to minimize the difference between $\mathbf{o}$ and the labels $\mathbf{y}$. While it turns out that treating classification as a vector-valued regression problem works surprisingly well, it is nonetheless unsatisfactory in the following ways:

- There is no guarantee that the outputs o_i sum up to 1 in the way we expect probabilities to behave.
- There is no guarantee that the outputs o_i are even nonnegative, even if their outputs sum up to 1, or that they do not exceed 1.

Both aspects render the estimation problem difficult to solve and the solution very brittle to outliers. For instance, if we assume that there is a positive linear dependency between the number of bedrooms and the likelihood that someone will buy a house, the probability

might exceed 1 when it comes to buying a mansion! As such, we need a mechanism to "squish" the outputs.

There are many ways we might accomplish this goal. For instance, we could assume that the outputs $\mathbf{o}$ are corrupted versions of $\mathbf{y}$, where the corruption occurs by means of adding noise $\boldsymbol{\epsilon}$ drawn from a normal distribution. In other words, $\mathbf{y} = \mathbf{o} + \boldsymbol{\epsilon}$, where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. This is the so-called probit model[88], first introduced by Fechner (1860). While appealing, it does not work quite as well nor lead to a particularly nice optimization problem, when compared to the softmax.

[88] 

Another way to accomplish this goal (and to ensure nonnegativity) is to use an exponential function $P(y = i) \propto \exp o_i$. This does indeed satisfy the requirement that the conditional class probability increases with increasing o_i, it is monotonic, and all probabilities are nonnegative. We can then transform these values so that they add up to 1 by dividing each by their sum. This process is called *normalization*. Putting these two pieces together gives us the *softmax* function:

$$\hat{\mathbf{y}} = \operatorname{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}. \tag{4.1.3}$$

Note that the largest coordinate of $\mathbf{o}$ corresponds to the most likely class according to $\hat{\mathbf{y}}$. Moreover, because the softmax operation preserves the ordering among its arguments, we do not need to compute the softmax to determine which class has been assigned the highest probability. Thus,

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j. \tag{4.1.4}$$

The idea of a softmax dates back to Gibbs (1902), who adapted ideas from physics. Dating even further back, Boltzmann, the father of modern statistical physics, used this trick to model a distribution over energy states in gas molecules. In particular, he discovered that the prevalence of a state of energy in a thermodynamic ensemble, such as the molecules in a gas, is proportional to $\exp(-E/kT)$. Here, E is the energy of a state, T is the temperature, and k is the Boltzmann constant. When statisticians talk about increasing or decreasing the "temperature" of a statistical system, they refer to changing T in order to favor lower or higher energy states. Following Gibbs' idea, energy equates to error. Energy-based models (Ranzato *et al.*, 2007) use this point of view when describing problems in deep learning.

Vectorization

To improve computational efficiency, we vectorize calculations in minibatches of data. Assume that we are given a minibatch $\mathbf{X} \in \mathbb{R}^{n \times d}$ of n examples with dimensionality (number of inputs) d. Moreover, assume that we have q categories in the output. Then the weights satisfy $\mathbf{W} \in \mathbb{R}^{d \times q}$ and the bias satisfies $\mathbf{b} \in \mathbb{R}^{1 \times q}$.

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \operatorname{softmax}(\mathbf{O}). \end{aligned} \tag{4.1.5}$$

This accelerates the dominant operation into a matrix–matrix product $\mathbf{XW}$. Moreover, since each row in $\mathbf{X}$ represents a data example, the softmax operation itself can be computed *rowwise*: for each row of $\mathbf{O}$, exponentiate all entries and then normalize them by the sum. Note, though, that care must be taken to avoid exponentiating and taking logarithms of large numbers, since this can cause numerical overflow or underflow. Deep learning frameworks take care of this automatically.

4.1.2 Loss Function

Now that we have a mapping from features $\mathbf{x}$ to probabilities $\hat{\mathbf{y}}$, we need a way to optimize the accuracy of this mapping. We will rely on maximum likelihood estimation, the very same method that we encountered when providing a probabilistic justification for the mean squared error loss in Section 3.1.3.

Log-Likelihood

The softmax function gives us a vector $\hat{\mathbf{y}}$, which we can interpret as the (estimated) conditional probabilities of each class, given any input $\mathbf{x}$, such as $\hat{y}_1 = P(y = \text{cat} \mid \mathbf{x})$. In the following we assume that for a dataset with features $\mathbf{X}$ the labels $\mathbf{Y}$ are represented using a one-hot encoding label vector. We can compare the estimates with reality by checking how probable the actual classes are according to our model, given the features:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}). \tag{4.1.6}$$

We are allowed to use the factorization since we assume that each label is drawn independently from its respective distribution $P(\mathbf{y} \mid \mathbf{x}^{(i)})$. Since maximizing the product of terms is awkward, we take the negative logarithm to obtain the equivalent problem of minimizing the negative log-likelihood:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \tag{4.1.7}$$

where for any pair of label $\mathbf{y}$ and model prediction $\hat{\mathbf{y}}$ over q classes, the loss function l is

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j. \tag{4.1.8}$$

For reasons explained later on, the loss function in $(4.1.8)$ is commonly called the *cross-entropy loss*. Since $\mathbf{y}$ is a one-hot vector of length q, the sum over all its coordinates j vanishes for all but one term. Note that the loss $l(\mathbf{y}, \hat{\mathbf{y}})$ is bounded from below by 0 whenever $\hat{\mathbf{y}}$ is a probability vector: no single entry is larger than 1, hence their negative logarithm cannot be lower than 0; $l(\mathbf{y}, \hat{\mathbf{y}}) = 0$ only if we predict the actual label with *certainty*. This can never happen for any finite setting of the weights because taking a softmax output towards 1 requires taking the corresponding input o_i to infinity (or all other outputs o_j for $j \neq i$ to negative infinity). Even if our model could assign an output probability of 0, any error made when assigning such high confidence would incur infinite loss ($-\log 0 = \infty$).

Softmax and Cross-Entropy Loss

Since the softmax function and the corresponding cross-entropy loss are so common, it is worth understanding a bit better how they are computed. Plugging (4.1.3) into the definition of the loss in (4.1.8) and using the definition of the softmax we obtain

$$\begin{aligned} l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} \\ &= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \\ &= \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j. \end{aligned} \tag{4.1.9}$$

To understand a bit better what is going on, consider the derivative with respect to any logit o_j. We get

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j. \tag{4.1.10}$$

In other words, the derivative is the difference between the probability assigned by our model, as expressed by the softmax operation, and what actually happened, as expressed by elements in the one-hot label vector. In this sense, it is very similar to what we saw in regression, where the gradient was the difference between the observation y and estimate $\hat{y}$. This is not a coincidence. In any exponential family model, the gradients of the log-likelihood are given by precisely this term. This fact makes computing gradients easy in practice.

Now consider the case where we observe not just a single outcome but an entire distribution over outcomes. We can use the same representation as before for the label $\mathbf{y}$. The only difference is that rather than a vector containing only binary entries, say $(0, 0, 1)$, we now have a generic probability vector, say $(0.1, 0.2, 0.7)$. The math that we used previously to define the loss l in (4.1.8) still works well, just that the interpretation is slightly more general. It is the expected value of the loss for a distribution over labels. This loss is called the *cross-entropy loss* and it is one of the most commonly used losses for classification problems. We can demystify the name by introducing just the basics of information theory. In a nutshell, it measures the number of bits needed to encode what we see, $\mathbf{y}$, relative to what we predict that should happen, $\hat{\mathbf{y}}$. We provide a very basic explanation in the following. For further details on information theory see Cover and Thomas (1999) or MacKay (2003).

4.1.3 Information Theory Basics

Many deep learning papers use intuition and terms from information theory. To make sense of them, we need some common language. This is a survival guide. *Information theory* deals with the problem of encoding, decoding, transmitting, and manipulating information (also known as data).

Entropy

The central idea in information theory is to quantify the amount of information contained in data. This places a limit on our ability to compress data. For a distribution P its *entropy*, $H[P]$, is defined as:

$$H[P] = \sum_j -P(j) \log P(j). \tag{4.1.11}$$

One of the fundamental theorems of information theory states that in order to encode data drawn randomly from the distribution P, we need at least $H[P]$ "nats" to encode it (Shannon, 1948). If you wonder what a "nat" is, it is the equivalent of bit but when using a code with base e rather than one with base 2. Thus, one nat is $\frac{1}{\log(2)} \approx 1.44$ bit.

Surprisal

You might be wondering what compression has to do with prediction. Imagine that we have a stream of data that we want to compress. If it is always easy for us to predict the next token, then this data is easy to compress. Take the extreme example where every token in the stream always takes the same value. That is a very boring data stream! And not only it is boring, but it is also easy to predict. Because the tokens are always the same, we do not have to transmit any information to communicate the contents of the stream. Easy to predict, easy to compress.

However if we cannot perfectly predict every event, then we might sometimes be surprised. Our surprise is greater when an event is assigned lower probability. Claude Shannon settled on $\log \frac{1}{P(j)} = -\log P(j)$ to quantify one's *surprisal* at observing an event j having assigned it a (subjective) probability $P(j)$. The entropy defined in $(4.1.11)$ is then the *expected surprisal* when one assigned the correct probabilities that truly match the data-generating process.

Cross-Entropy Revisited

So if entropy is the level of surprise experienced by someone who knows the true probability, then you might be wondering, what is cross-entropy? The cross-entropy *from P to Q*, denoted $H(P, Q)$, is the expected surprisal of an observer with subjective probabilities Q upon seeing data that was actually generated according to probabilities P. This is given by $H(P, Q) \stackrel{\text{def}}{=} \sum_j -P(j) \log Q(j)$. The lowest possible cross-entropy is achieved when $P = Q$. In this case, the cross-entropy from P to Q is $H(P, P) = H(P)$.

In short, we can think of the cross-entropy classification objective in two ways: (i) as maximizing the likelihood of the observed data; and (ii) as minimizing our surprisal (and thus the number of bits) required to communicate the labels.

4.1.4 Summary and Discussion

In this section, we encountered the first nontrivial loss function, allowing us to optimize over *discrete* output spaces. Key in its design was that we took a probabilistic approach, treating

discrete categories as instances of draws from a probability distribution. As a side effect, we encountered the softmax, a convenient activation function that transforms outputs of an ordinary neural network layer into valid discrete probability distributions. We saw that the derivative of the cross-entropy loss when combined with softmax behaves very similarly to the derivative of squared error; namely by taking the difference between the expected behavior and its prediction. And, while we were only able to scratch the very surface of it, we encountered exciting connections to statistical physics and information theory.

While this is enough to get you on your way, and hopefully enough to whet your appetite, we hardly dived deep here. Among other things, we skipped over computational considerations. Specifically, for any fully connected layer with d inputs and q outputs, the parametrization and computational cost is $O(dq)$, which can be prohibitively high in practice. Fortunately, this cost of transforming d inputs into q outputs can be reduced through approximation and compression. For instance Deep Fried Convnets (Yang *et al.*, 2015) uses a combination of permutations, Fourier transforms, and scaling to reduce the cost from quadratic to log-linear. Similar techniques work for more advanced structural matrix approximations (Sindhwani *et al.*, 2015). Lastly, we can use quaternion-like decompositions to reduce the cost to $O(\frac{dq}{n})$, again if we are willing to trade off a small amount of accuracy for computational and storage cost (Zhang *et al.*, 2021) based on a compression factor n. This is an active area of research. What makes it challenging is that we do not necessarily strive for the most compact representation or the smallest number of floating point operations but rather for the solution that can be executed most efficiently on modern GPUs.

4.1.5 Exercises

1. We can explore the connection between exponential families and softmax in some more depth.
   1. Compute the second derivative of the cross-entropy loss $l(\mathbf{y}, \hat{\mathbf{y}})$ for softmax.
   2. Compute the variance of the distribution given by $\mathrm{softmax}(\mathbf{o})$ and show that it matches the second derivative computed above.
2. Assume that we have three classes which occur with equal probability, i.e., the probability vector is $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
   1. What is the problem if we try to design a binary code for it?
   2. Can you design a better code? Hint: what happens if we try to encode two independent observations? What if we encode n observations jointly?
3. When encoding signals transmitted over a physical wire, engineers do not always use binary codes. For instance, PAM-3[89] uses three signal levels $\{-1, 0, 1\}$ as opposed to two levels $\{0, 1\}$. How many ternary units do you need to transmit an integer in the range $\{0, \ldots, 7\}$? Why might this be a better idea in terms of electronics?
4. The Bradley–Terry model[90] uses a logistic model to capture preferences. For a user to

choose between apples and oranges one assumes scores o_{apple} and o_{orange}. Our requirements are that larger scores should lead to a higher likelihood in choosing the associated item and that the item with the largest score is the most likely one to be chosen (Bradley and Terry, 1952).

1. Prove that softmax satisfies this requirement.
2. What happens if you want to allow for a default option of choosing neither apples nor oranges? Hint: now the user has three choices.

5. Softmax gets its name from the following mapping: $\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$.
   1. Prove that $\text{RealSoftMax}(a, b) > \max(a, b)$.
   2. How small can you make the difference between both functions? Hint: without loss of generality you can set $b = 0$ and $a \geq b$.
   3. Prove that this holds for $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b)$, provided that $\lambda > 0$.
   4. Show that for $\lambda \to \infty$ we have $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$.
   5. Construct an analogous softmin function.
   6. Extend this to more than two numbers.
6. The function $g(\mathbf{x}) \stackrel{\text{def}}{=} \log \sum_i \exp x_i$ is sometimes also referred to as the log-partition function[91].
   1. Prove that the function is convex. Hint: to do so, use the fact that the first derivative amounts to the probabilities from the softmax function and show that the second derivative is the variance.
   2. Show that g is translation invariant, i.e., $g(\mathbf{x} + b) = g(\mathbf{x})$.
   3. What happens if some of the coordinates x_i are very large? What happens if they're all very small?
   4. Show that if we choose $b = \max_i x_i$ we end up with a numerically stable implementation.
7. Assume that we have some probability distribution P. Suppose we pick another distribution Q with $Q(i) \propto P(i)^\alpha$ for $\alpha > 0$.
   1. Which choice of α corresponds to doubling the temperature? Which choice corresponds to halving it?
   2. What happens if we let the temperature approach 0?
   3. What happens if we let the temperature approach ∞?

[91]

[92]

Discussions[92].

4.2 The Image Classification Dataset

One widely used dataset for image classification is the MNIST dataset[93] (LeCun *et al.*, 1998) of handwritten digits. At the time of its release in the 1990s it posed a formidable challenge to most machine learning algorithms, consisting of 60,000 images of 28×28 pixels resolution (plus a test dataset of 10,000 images). To put things into perspective, back in 1995, a Sun SPARCStation 5 with a whopping 64MB of RAM and a blistering 5 MFLOPs was considered state of the art equipment for machine learning at AT&T Bell Laboratories. Achieving high accuracy on digit recognition was a key component in automating letter sorting for the USPS in the 1990s. Deep networks such as LeNet-5 (LeCun *et al.*, 1995), support vector machines with invariances (Schölkopf *et al.*, 1996), and tangent distance classifiers (Simard *et al.*, 1998) all could reach error rates below 1%.

For over a decade, MNIST served as *the* point of reference for comparing machine learning algorithms. While it had a good run as a benchmark dataset, even simple models by today's standards achieve classification accuracy over 95%, making it unsuitable for distinguishing between strong models and weaker ones. Even more, the dataset allows for *very* high levels of accuracy, not typically seen in many classification problems. This skewed algorithmic development towards specific families of algorithms that can take advantage of clean datasets, such as active set methods and boundary-seeking active set algorithms. Today, MNIST serves as more of a sanity check than as a benchmark. ImageNet (Deng *et al.*, 2009) poses a much more relevant challenge. Unfortunately, ImageNet is too large for many of the examples and illustrations in this book, as it would take too long to train to make the examples interactive. As a substitute we will focus our discussion in the coming sections on the qualitatively similar, but much smaller Fashion-MNIST dataset (Xiao *et al.*, 2017) which was released in 2017. It contains images of 10 categories of clothing at 28×28 pixels resolution.

```
%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

4.2.1 Loading the Dataset

Since the Fashion-MNIST dataset is so useful, all major frameworks provide preprocessed versions of it. We can download and read it into memory using built-in framework utilities.

```
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)
```

Fashion-MNIST consists of images from 10 categories, each represented by 6000 images in the training dataset and by 1000 in the test dataset. A *test dataset* is used for evaluating model performance (it must not be used for training). Consequently the training set and the test set contain 60,000 and 10,000 images, respectively.

```
data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

```
(60000, 10000)
```

The images are grayscale and upscaled to 32×32 pixels in resolution above. This is similar to the original MNIST dataset which consisted of (binary) black and white images. Note, though, that most modern image data has three channels (red, green, blue) and that hyperspectral images can have in excess of 100 channels (the HyMap sensor has 126 channels). By convention we store an image as a $c \times h \times w$ tensor, where c is the number of color channels, h is the height and w is the width.

```
data.train[0][0].shape
```

```
torch.Size([1, 32, 32])
```

The categories of Fashion-MNIST have human-understandable names. The following convenience method converts between numeric labels and their names.

```
@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]
```

4.2.2 Reading a Minibatch

To make our life easier when reading from the training and test sets, we use the built-in data iterator rather than creating one from scratch. Recall that at each iteration, a data iterator

reads a minibatch of data with size `batch_size`. We also randomly shuffle the examples for the training data iterator.

```
@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)
```

To see how this works, let's load a minibatch of images by invoking the `train_dataloader` method. It contains 64 images.

```
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
```

Let's look at the time it takes to read the images. Even though it is a built-in loader, it is not blazingly fast. Nonetheless, this is sufficient since processing images with a deep network takes quite a bit longer. Hence it is good enough that training a network will not be I/O constrained.

```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

```
'4.69 sec'
```

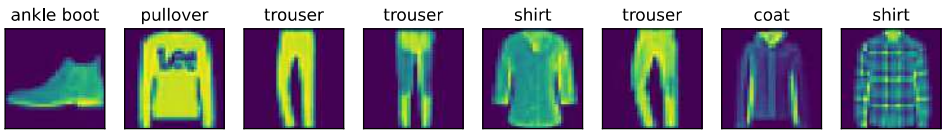
4.2.3 Visualization

We will often be using the Fashion-MNIST dataset. A convenience function `show_images` can be used to visualize the images and the associated labels. Skipping implementation details, we just show the interface below: we only need to know how to invoke `d2l.show_images` rather than how it works for such utility functions.

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """Plot a list of images."""
    raise NotImplementedError
```

Let's put it to good use. In general, it is a good idea to visualize and inspect data that you are training on. Humans are very good at spotting oddities and because of that, visualization serves as an additional safeguard against mistakes and errors in the design of experiments. Here are the images and their corresponding labels (in text) for the first few examples in the training dataset.

```
@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)
    d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels)
batch = next(iter(data.val_dataloader()))
data.visualize(batch)
```



We are now ready to work with the Fashion-MNIST dataset in the sections that follow.

4.2.4 Summary

We now have a slightly more realistic dataset to use for classification. Fashion-MNIST is an apparel classification dataset consisting of images representing 10 categories. We will use this dataset in subsequent sections and chapters to evaluate various network designs, from a simple linear model to advanced residual networks. As we commonly do with images, we read them as a tensor of shape (batch size, number of channels, height, width). For now, we only have one channel as the images are grayscale (the visualization above uses a false color palette for improved visibility).

Lastly, data iterators are a key component for efficient performance. For instance, we might use GPUs for efficient image decompression, video transcoding, or other preprocessing. Whenever possible, you should rely on well-implemented data iterators that exploit high-performance computing to avoid slowing down your training loop.

4.2.5 Exercises

1. Does reducing the `batch_size` (for instance, to 1) affect the reading performance?

2. The data iterator performance is important. Do you think the current implementation is fast enough? Explore various options to improve it. Use a system profiler to find out where the bottlenecks are.

94 

3. Check out the framework's online API documentation. Which other datasets are available?

Discussions[94].

4.3 The Base Classification Model

You may have noticed that the implementations from scratch and the concise implementation using framework functionality were quite similar in the case of regression. The same is true for classification. Since many models in this book deal with classification, it is worth adding functionalities to support this setting specifically. This section provides a base class for classification models to simplify future code.

```
import torch
from d2l import torch as d2l
```

4.3.1 The `Classifier` Class

We define the `Classifier` class below. In the `validation_step` we report both the loss value and the classification accuracy on a validation batch. We draw an update for every `num_val_batches` batches. This has the benefit of generating the averaged loss and accuracy on the whole validation data. These average numbers are not exactly correct if the final batch contains fewer examples, but we ignore this minor difference to keep the code simple.

```
class Classifier(d2l.Module):  #@save
    """The base class of classification models."""
    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)
```

By default we use a stochastic gradient descent optimizer, operating on minibatches, just as we did in the context of linear regression.

```
@d2l.add_to_class(d2l.Module)  #@save
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

4.3.2 Accuracy

Given the predicted probability distribution `y_hat`, we typically choose the class with the highest predicted probability whenever we must output a hard prediction. Indeed, many applications require that we make a choice. For instance, Gmail must categorize an email into “Primary”, “Social”, “Updates”, “Forums”, or “Spam”. It might estimate probabilities internally, but at the end of the day it has to choose one among the classes.

When predictions are consistent with the label class `y`, they are correct. The classification accuracy is the fraction of all predictions that are correct. Although it can be difficult to optimize accuracy directly (it is not differentiable), it is often the performance measure that

we care about the most. It is often *the* relevant quantity in benchmarks. As such, we will nearly always report it when training classifiers.

Accuracy is computed as follows. First, if `y_hat` is a matrix, we assume that the second dimension stores prediction scores for each class. We use `argmax` to obtain the predicted class by the index for the largest entry in each row. Then we compare the predicted class with the ground truth `y` elementwise. Since the equality operator `==` is sensitive to data types, we convert `y_hat`'s data type to match that of `y`. The result is a tensor containing entries of 0 (false) and 1 (true). Taking the sum yields the number of correct predictions.

```
@d2l.add_to_class(Classifier)  #@save
def accuracy(self, Y_hat, Y, averaged=True):
    """Compute the number of correct predictions."""
    Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1]))
    preds = Y_hat.argmax(axis=1).type(Y.dtype)
    compare = (preds == Y.reshape(-1)).type(torch.float32)
    return compare.mean() if averaged else compare
```

4.3.3 Summary

Classification is a sufficiently common problem that it warrants its own convenience functions. Of central importance in classification is the *accuracy* of the classifier. Note that while we often care primarily about accuracy, we train classifiers to optimize a variety of other objectives for statistical and computational reasons. However, regardless of which loss function was minimized during training, it is useful to have a convenience method for assessing the accuracy of our classifier empirically.

4.3.4 Exercises

1. Denote by L_v the validation loss, and let L_v^q be its quick and dirty estimate computed by the loss function averaging in this section. Lastly, denote by l_v^b the loss on the last minibatch. Express L_v in terms of L_v^q, l_v^b, and the sample and minibatch sizes.

95 

2. Show that the quick and dirty estimate L_v^q is unbiased. That is, show that $E[L_v] = E[L_v^q]$. Why would you still want to use L_v instead?

3. Given a multiclass classification loss, denoting by $l(y, y')$ the penalty of estimating y' when we see y and given a probabilty $p(y \mid x)$, formulate the rule for an optimal selection of y'. Hint: express the expected loss, using l and $p(y \mid x)$.

Discussions[95].

4.4 Softmax Regression Implementation from Scratch

Because softmax regression is so fundamental, we believe that you ought to know how to implement it yourself. Here, we limit ourselves to defining the softmax-specific aspects of the model and reuse the other components from our linear regression section, including the training loop.

```
import torch
from d2l import torch as d2l
```

4.4.1 The Softmax

Let's begin with the most important part: the mapping from scalars to probabilities. For a refresher, recall the operation of the sum operator along specific dimensions in a tensor, as discussed in Section 2.3.6 and Section 2.3.7. Given a matrix `X` we can sum over all elements (by default) or only over elements in the same axis. The `axis` variable lets us compute row and column sums:

```
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdims=True), X.sum(1, keepdims=True)
```

```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

Computing the softmax requires three steps: (i) exponentiation of each term; (ii) a sum over each row to compute the normalization constant for each example; (iii) division of each row by its normalization constant, ensuring that the result sums to 1:

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}. \tag{4.4.1}$$

The (logarithm of the) denominator is called the (log) *partition function*. It was introduced in statistical physics[96] to sum over all possible states in a thermodynamic ensemble. The implementation is straightforward:

```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition  # The broadcasting mechanism is applied here
```

For any input `X`, we turn each element into a nonnegative number. Each row sums up to 1, as is required for a probability. Caution: the code above is *not* robust against very large or very small arguments. While it is sufficient to illustrate what is happening, you should

not use this code verbatim for any serious purpose. Deep learning frameworks have such protections built in and we will be using the built-in softmax going forward.

```
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

```
(tensor([[0.2511, 0.1417, 0.1158, 0.2529, 0.2385],
         [0.2004, 0.1419, 0.1957, 0.2504, 0.2117]]),
 tensor([1., 1.]))
```

4.4.2 The Model

We now have everything that we need to implement the softmax regression model. As in our linear regression example, each instance will be represented by a fixed-length vector. Since the raw data here consists of 28×28 pixel images, we flatten each image, treating them as vectors of length 784. In later chapters, we will introduce convolutional neural networks, which exploit the spatial structure in a more satisfying way.

In softmax regression, the number of outputs from our network should be equal to the number of classes. Since our dataset has 10 classes, our network has an output dimension of 10. Consequently, our weights constitute a 784×10 matrix plus a 1×10 row vector for the biases. As with linear regression, we initialize the weights `W` with Gaussian noise. The biases are initialized as zeros.

```
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]
```

The code below defines how the network maps each input to an output. Note that we flatten each 28×28 pixel image in the batch into a vector using `reshape` before passing the data through our model.

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = X.reshape((-1, self.W.shape[0]))
    return softmax(torch.matmul(X, self.W) + self.b)
```

4.4.3 The Cross-Entropy Loss

Next we need to implement the cross-entropy loss function (introduced in Section 4.1.2). This may be the most common loss function in all of deep learning. At the moment, appli-

cations of deep learning easily cast as classification problems far outnumber those better treated as regression problems.

Recall that cross-entropy takes the negative log-likelihood of the predicted probability assigned to the true label. For efficiency we avoid Python for-loops and use indexing instead. In particular, the one-hot encoding in $\mathbf{y}$ allows us to select the matching terms in $\hat{\mathbf{y}}$.

To see this in action we create sample data `y_hat` with 2 examples of predicted probabilities over 3 classes and their corresponding labels `y`. The correct labels are 0 and 2 respectively (i.e., the first and third class). Using `y` as the indices of the probabilities in `y_hat`, we can pick out terms efficiently.

```
y = torch.tensor([0, 2])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
tensor([0.1000, 0.5000])
```

Now we can implement the cross-entropy loss function by averaging over the logarithms of the selected probabilities.

```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

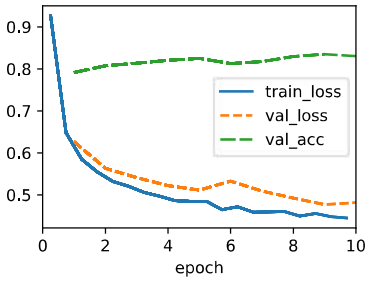
```
tensor(1.4979)
```

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

4.4.4 Training

We reuse the `fit` method defined in Section 3.4 to train the model with 10 epochs. Note that the number of epochs (`max_epochs`), the minibatch size (`batch_size`), and learning rate (`lr`) are adjustable hyperparameters. That means that while these values are not learned during our primary training loop, they still influence the performance of our model, both vis-à-vis training and generalization performance. In practice you will want to choose these values based on the *validation* split of the data and then, ultimately, to evaluate your final model on the *test* split. As discussed in Section 3.6.3, we will regard the test data of Fashion-MNIST as the validation set, thus reporting validation loss and validation accuracy on this split.

```
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)
```



4.4.5 Prediction

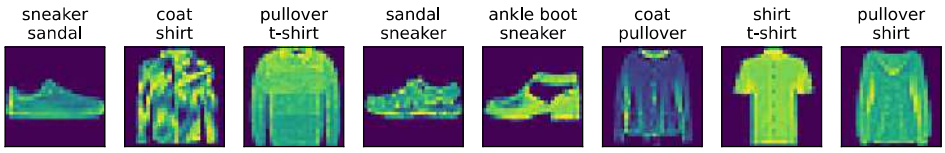
Now that training is complete, our model is ready to classify some images.

```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

```
torch.Size([256])
```

We are more interested in the images we label *incorrectly*. We visualize them by comparing their actual labels (first line of text output) with the predictions from the model (second line of text output).

```
wrong = preds.type(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)
```



4.4.6 Summary

By now we are starting to get some experience with solving linear regression and classification problems. With it, we have reached what would arguably be the state of the art of

1960–1970s of statistical modeling. In the next section, we will show you how to leverage deep learning frameworks to implement this model much more efficiently.

4.4.7 Exercises

1. In this section, we directly implemented the softmax function based on the mathematical definition of the softmax operation. As discussed in Section 4.1 this can cause numerical instabilities.
   1. Test whether `softmax` still works correctly if an input has a value of 100.
   2. Test whether `softmax` still works correctly if the largest of all inputs is smaller than −100?
   3. Implement a fix by looking at the value relative to the largest entry in the argument.
2. Implement a `cross_entropy` function that follows the definition of the cross-entropy loss function $\sum_i y_i \log \hat{y}_i$.
   1. Try it out in the code example of this section.
   2. Why do you think it runs more slowly?
   3. Should you use it? When would it make sense to?
   4. What do you need to be careful of? Hint: consider the domain of the logarithm.
3. Is it always a good idea to return the most likely label? For example, would you do this for medical diagnosis? How would you try to address this?
4. Assume that we want to use softmax regression to predict the next word based on some features. What are some problems that might arise from a large vocabulary?
5. Experiment with the hyperparameters of the code in this section. In particular:
   1. Plot how the validation loss changes as you change the learning rate.
   2. Do the validation and training loss change as you change the minibatch size? How large or small do you need to go before you see an effect?


97

Discussions[97].

4.5 Concise Implementation of Softmax Regression

Just as high-level deep learning frameworks made it easier to implement linear regression (see Section 3.5), they are similarly convenient here.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

4.5.1 Defining the Model

As in Section 3.5, we construct our fully connected layer using the built-in layer. The built-in `__call__` method then invokes `forward` whenever we need to apply the network to some input.

We use a `Flatten` layer to convert the fourth-order tensor `X` to second order by keeping the dimensionality along the first axis unchanged.

```
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)
```

4.5.2 Softmax Revisited

In Section 4.4 we calculated our model's output and applied the cross-entropy loss. While this is perfectly reasonable mathematically, it is risky computationally, because of numerical underflow and overflow in the exponentiation.

Recall that the softmax function computes probabilities via $\hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$. If some of the o_k are very large, i.e., very positive, then $\exp(o_k)$ might be larger than the largest number we can have for certain data types. This is called *overflow*. Likewise, if every argument is a very large negative number, we will get *underflow*. For instance, single precision floating point numbers approximately cover the range of 10^{-38} to 10^{38}. As such, if the largest term in $\mathbf{o}$ lies outside the interval $[-90, 90]$, the result will not be stable. A way round this problem is to subtract $\bar{o} \stackrel{\text{def}}{=} \max_k o_k$ from all entries:

$$\hat{y}_j = \frac{\exp o_j}{\sum_k \exp o_k} = \frac{\exp(o_j - \bar{o}) \exp \bar{o}}{\sum_k \exp(o_k - \bar{o}) \exp \bar{o}} = \frac{\exp(o_j - \bar{o})}{\sum_k \exp(o_k - \bar{o})}. \tag{4.5.1}$$

By construction we know that $o_j - \bar{o} \leq 0$ for all j. As such, for a q-class classification problem, the denominator is contained in the interval $[1, q]$. Moreover, the numerator never exceeds 1, thus preventing numerical overflow. Numerical underflow only occurs when $\exp(o_j - \bar{o})$ numerically evaluates as 0. Nonetheless, a few steps down the road we might find ourselves in trouble when we want to compute $\log \hat{y}_j$ as $\log 0$. In particular, in backpropagation, we might find ourselves faced with a screenful of the dreaded `NaN` (Not a Number) results.

Fortunately, we are saved by the fact that even though we are computing exponential functions, we ultimately intend to take their log (when calculating the cross-entropy loss). By combining softmax and cross-entropy, we can escape the numerical stability issues altogether. We have:

$$\log \hat{y}_j = \log \frac{\exp(o_j - \bar{o})}{\sum_k \exp(o_k - \bar{o})} = o_j - \bar{o} - \log \sum_k \exp(o_k - \bar{o}). \tag{4.5.2}$$

This avoids both overflow and underflow. We will want to keep the conventional softmax function handy in case we ever want to evaluate the output probabilities by our model. But instead of passing softmax probabilities into our new loss function, we just pass the logits and compute the softmax and its log all at once inside the cross-entropy loss function, which does smart things like the "LogSumExp trick"[98].

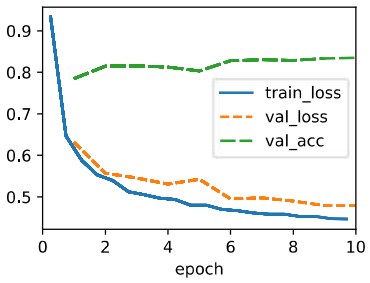
98 

```
@d2l.add_to_class(d2l.Classifier)  #@save
def loss(self, Y_hat, Y, averaged=True):
    Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1]))
    Y = Y.reshape((-1,))
    return F.cross_entropy(
        Y_hat, Y, reduction='mean' if averaged else 'none')
```

4.5.3 Training

Next we train our model. We use Fashion-MNIST images, flattened to 784-dimensional feature vectors.

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



As before, this algorithm converges to a solution that is reasonably accurate, albeit this time with fewer lines of code than before.

4.5.4 Summary

High-level APIs are very convenient at hiding from their user potentially dangerous aspects, such as numerical stability. Moreover, they allow users to design models concisely with

very few lines of code. This is both a blessing and a curse. The obvious benefit is that it makes things highly accessible, even to engineers who never took a single class of statistics in their life (in fact, they are part of the target audience of the book). But hiding the sharp edges also comes with a price: a disincentive to add new and different components on your own, since there is little muscle memory for doing it. Moreover, it makes it more difficult to *fix* things whenever the protective padding of a framework fails to cover all the corner cases entirely. Again, this is due to lack of familiarity.

As such, we strongly urge you to review *both* the bare bones and the elegant versions of many of the implementations that follow. While we emphasize ease of understanding, the implementations are nonetheless usually quite performant (convolutions are the big exception here). It is our intention to allow you to build on these when you invent something new that no framework can give you.

4.5.5 Exercises

1. Deep learning uses many different number formats, including FP64 double precision (used extremely rarely), FP32 single precision, BFLOAT16 (good for compressed representations), FP16 (very unstable), TF32 (a new format from NVIDIA), and INT8. Compute the smallest and largest argument of the exponential function for which the result does not lead to numerical underflow or overflow.
2. INT8 is a very limited format consisting of nonzero numbers from 1 to 255. How could you extend its dynamic range without using more bits? Do standard multiplication and addition still work?
3. Increase the number of epochs for training. Why might the validation accuracy decrease after a while? How could we fix this?
4. What happens as you increase the learning rate? Compare the loss curves for several learning rates. Which one works better? When?

Discussions[99].

99

4.6 Generalization in Classification

So far, we have focused on how to tackle multiclass classification problems by training (linear) neural networks with multiple outputs and softmax functions. Interpreting our model's outputs as probabilistic predictions, we motivated and derived the cross-entropy loss function, which calculates the negative log likelihood that our model (for a fixed set of parameters) assigns to the actual labels. And finally, we put these tools into practice by fitting our model to the training set. However, as always, our goal is to learn *general patterns*, as assessed empirically on previously unseen data (the test set). High accuracy on the training set means nothing. Whenever each of our inputs is unique (and indeed this is true for most high-dimensional datasets), we can attain perfect accuracy on the training

set by just memorizing the dataset on the first training epoch, and subsequently looking up the label whenever we see a new image. And yet, memorizing the exact labels associated with the exact training examples does not tell us how to classify new examples. Absent further guidance, we might have to fall back on random guessing whenever we encounter new examples.

A number of burning questions demand immediate attention:

1. How many test examples do we need to give a good estimate of the accuracy of our classifiers on the underlying population?
2. What happens if we keep evaluating models on the same test repeatedly?
3. Why should we expect that fitting our linear models to the training set should fare any better than our naive memorization scheme?

Whereas Section 3.6 introduced the basics of overfitting and generalization in the context of linear regression, this chapter will go a little deeper, introducing some of the foundational ideas of statistical learning theory. It turns out that we often can guarantee generalization *a priori*: for many models, and for any desired upper bound on the generalization gap ϵ, we can often determine some required number of samples n such that if our training set contains at least n samples, our empirical error will lie within ϵ of the true error, *for any data generating distribution*. Unfortunately, it also turns out that while these sorts of guarantees provide a profound set of intellectual building blocks, they are of limited practical utility to the deep learning practitioner. In short, these guarantees suggest that ensuring generalization of deep neural networks *a priori* requires an absurd number of examples (perhaps trillions or more), even when we find that, on the tasks we care about, deep neural networks typically generalize remarkably well with far fewer examples (thousands). Thus deep learning practitioners often forgo *a priori* guarantees altogether, instead employing methods that have generalized well on similar problems in the past, and certifying generalization *post hoc* through empirical evaluations. When we get to Chapter 5, we will revisit generalization and provide a light introduction to the vast scientific literature that has sprung in attempts to explain why deep neural networks generalize in practice.

4.6.1 The Test Set

Since we have already begun to rely on test sets as the gold standard method for assessing generalization error, let's get started by discussing the properties of such error estimates. Let's focus on a fixed classifier f, without worrying about how it was obtained. Moreover suppose that we possess a *fresh* dataset of examples $\mathcal{D} = (\mathbf{x}^{(i)}, y^{(i)})_{i=1}^{n}$ that were not used to train the classifier f. The *empirical error* of our classifier f on $\mathcal{D}$ is simply the fraction of instances for which the prediction $f(\mathbf{x}^{(i)})$ disagrees with the true label $y^{(i)}$, and is given by the following expression:

$$\epsilon_{\mathcal{D}}(f) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}(f(\mathbf{x}^{(i)}) \neq y^{(i)}). \tag{4.6.1}$$

By contrast, the *population error* is the *expected* fraction of examples in the underlying population (some distribution $P(X, Y)$ characterized by probability density function $p(\mathbf{x}, y)$)

for which our classifier disagrees with the true label:

$$\epsilon(f) = E_{(\mathbf{x},y)\sim P}\mathbf{1}(f(\mathbf{x}) \neq y) = \int\int \mathbf{1}(f(\mathbf{x}) \neq y)p(\mathbf{x}, y)\, d\mathbf{x}dy. \tag{4.6.2}$$

While $\epsilon(f)$ is the quantity that we actually care about, we cannot observe it directly, just as we cannot directly observe the average height in a large population without measuring every single person. We can only estimate this quantity based on samples. Because our test set $\mathcal{D}$ is statistically representative of the underlying population, we can view $\epsilon_{\mathcal{D}}(f)$ as a statistical estimator of the population error $\epsilon(f)$. Moreover, because our quantity of interest $\epsilon(f)$ is an expectation (of the random variable $\mathbf{1}(f(X) \neq Y)$) and the corresponding estimator $\epsilon_{\mathcal{D}}(f)$ is the sample average, estimating the population error is simply the classic problem of mean estimation, which you may recall from Section 2.6.

An important classical result from probability theory called the *central limit theorem* guarantees that whenever we possess n random samples $a_1, ..., a_n$ drawn from any distribution with mean μ and standard deviation σ, then, as the number of samples n approaches infinity, the sample average $\hat{\mu}$ approximately tends towards a normal distribution centered at the true mean and with standard deviation $\sigma/\sqrt{n}$. Already, this tells us something important: as the number of examples grows large, our test error $\epsilon_{\mathcal{D}}(f)$ should approach the true error $\epsilon(f)$ at a rate of $O(1/\sqrt{n})$. Thus, to estimate our test error twice as precisely, we must collect four times as large a test set. To reduce our test error by a factor of one hundred, we must collect ten thousand times as large a test set. In general, such a rate of $O(1/\sqrt{n})$ is often the best we can hope for in statistics.

Now that we know something about the asymptotic rate at which our test error $\epsilon_{\mathcal{D}}(f)$ converges to the true error $\epsilon(f)$, we can zoom in on some important details. Recall that the random variable of interest $\mathbf{1}(f(X) \neq Y)$ can only take values 0 and 1 and thus is a Bernoulli random variable, characterized by a parameter indicating the probability that it takes value 1. Here, 1 means that our classifier made an error, so the parameter of our random variable is actually the true error rate $\epsilon(f)$. The variance σ^2 of a Bernoulli depends on its parameter (here, $\epsilon(f)$) according to the expression $\epsilon(f)(1 - \epsilon(f))$. While $\epsilon(f)$ is initially unknown, we know that it cannot be greater than 1. A little investigation of this function reveals that our variance is highest when the true error rate is close to 0.5 and can be far lower when it is close to 0 or close to 1. This tells us that the asymptotic standard deviation of our estimate $\epsilon_{\mathcal{D}}(f)$ of the error $\epsilon(f)$ (over the choice of the n test samples) cannot be any greater than $\sqrt{0.25/n}$.

If we ignore the fact that this rate characterizes behavior as the test set size approaches infinity rather than when we possess finite samples, this tells us that if we want our test error $\epsilon_{\mathcal{D}}(f)$ to approximate the population error $\epsilon(f)$ such that one standard deviation corresponds to an interval of ± 0.01, then we should collect roughly 2500 samples. If we want to fit two standard deviations in that range and thus be 95% confident that $\epsilon_{\mathcal{D}}(f) \in \epsilon(f) \pm 0.01$, then we will need 10,000 samples!

This turns out to be the size of the test sets for many popular benchmarks in machine learning. You might be surprised to find out that thousands of applied deep learning papers get published every year making a big deal out of error rate improvements of 0.01 or less. Of

course, when the error rates are much closer to 0, then an improvement of 0.01 can indeed be a big deal.

One pesky feature of our analysis thus far is that it really only tells us about asymptotics, i.e., how the relationship between $\epsilon_{\mathcal{D}}$ and ϵ evolves as our sample size goes to infinity. Fortunately, because our random variable is bounded, we can obtain valid finite sample bounds by applying an inequality due to Hoeffding (1963):

$$P(\epsilon_{\mathcal{D}}(f) - \epsilon(f) \geq t) < \exp\left(-2nt^2\right). \tag{4.6.3}$$

Solving for the smallest dataset size that would allow us to conclude with 95% confidence that the distance t between our estimate $\epsilon_{\mathcal{D}}(f)$ and the true error rate $\epsilon(f)$ does not exceed 0.01, you will find that roughly 15,000 examples are required as compared to the 10,000 examples suggested by the asymptotic analysis above. If you go deeper into statistics you will find that this trend holds generally. Guarantees that hold even in finite samples are typically slightly more conservative. Note that in the scheme of things, these numbers are not so far apart, reflecting the general usefulness of asymptotic analysis for giving us ballpark figures even if they are not guarantees we can take to court.

4.6.2 Test Set Reuse

In some sense, you are now set up to succeed at conducting empirical machine learning research. Nearly all practical models are developed and validated based on test set performance and you are now a master of the test set. For any fixed classifier f, you know how to evaluate its test error $\epsilon_{\mathcal{D}}(f)$, and know precisely what can (and cannot) be said about its population error $\epsilon(f)$.

So let's say that you take this knowledge and prepare to train your first model f_1. Knowing just how confident you need to be in the performance of your classifier's error rate you apply our analysis above to determine an appropriate number of examples to set aside for the test set. Moreover, let's assume that you took the lessons from Section 3.6 to heart and made sure to preserve the sanctity of the test set by conducting all of your preliminary analysis, hyperparameter tuning, and even selection among multiple competing model architectures on a validation set. Finally you evaluate your model f_1 on the test set and report an unbiased estimate of the population error with an associated confidence interval.

So far everything seems to be going well. However, that night you wake up at 3am with a brilliant idea for a new modeling approach. The next day, you code up your new model, tune its hyperparameters on the validation set and not only are you getting your new model f_2 to work but its error rate appears to be much lower than f_1's. However, the thrill of discovery suddenly fades as you prepare for the final evaluation. You do not have a test set!

Even though the original test set $\mathcal{D}$ is still sitting on your server, you now face two formidable problems. First, when you collected your test set, you determined the required level of precision under the assumption that you were evaluating a single classifier f. However, if you get into the business of evaluating multiple classifiers $f_1, ..., f_k$ on the same test set, you must consider the problem of false discovery. Before, you might have been 95% sure

that $\epsilon_{\mathcal{D}}(f) \in \epsilon(f) \pm 0.01$ for a single classifier f and thus the probability of a misleading result was a mere 5%. With k classifiers in the mix, it can be hard to guarantee that there is not even one among them whose test set performance is misleading. With 20 classifiers under consideration, you might have no power at all to rule out the possibility that at least one among them received a misleading score. This problem relates to multiple hypothesis testing, which despite a vast literature in statistics, remains a persistent problem plaguing scientific research.

If that is not enough to worry you, there is a special reason to distrust the results that you get on subsequent evaluations. Recall that our analysis of test set performance rested on the assumption that the classifier was chosen absent any contact with the test set and thus we could view the test set as drawn randomly from the underlying population. Here, not only are you testing multiple functions, the subsequent function f_2 was chosen after you observed the test set performance of f_1. Once information from the test set has leaked to the modeler, it can never be a true test set again in the strictest sense. This problem is called *adaptive overfitting* and has recently emerged as a topic of intense interest to learning theorists and statisticians (Dwork *et al.*, 2015). Fortunately, while it is possible to leak all information out of a holdout set, and the theoretical worst case scenarios are bleak, these analyses may be too conservative. In practice, take care to create real test sets, to consult them as infrequently as possible, to account for multiple hypothesis testing when reporting confidence intervals, and to dial up your vigilance more aggressively when the stakes are high and your dataset size is small. When running a series of benchmark challenges, it is often good practice to maintain several test sets so that after each round, the old test set can be demoted to a validation set.

4.6.3 Statistical Learning Theory

Put simply, *test sets are all that we really have*, and yet this fact seems strangely unsatisfying. First, we seldom possess a *true test set*—unless we are the ones creating the dataset, someone else has probably already evaluated their own classifier on our ostensible "test set". And even when we have first dibs, we soon find ourselves frustrated, wishing we could evaluate our subsequent modeling attempts without the gnawing feeling that we cannot trust our numbers. Moreover, even a true test set can only tell us *post hoc* whether a classifier has in fact generalized to the population, not whether we have any reason to expect *a priori* that it should generalize.

With these misgivings in mind, you might now be sufficiently primed to see the appeal of *statistical learning theory*, the mathematical subfield of machine learning whose practitioners aim to elucidate the fundamental principles that explain why/when models trained on empirical data can/will generalize to unseen data. One of the primary aims of statistical learning researchers has been to bound the generalization gap, relating the properties of the model class to the number of samples in the dataset.

Learning theorists aim to bound the difference between the *empirical error* $\epsilon_S(f_S)$ of a learned classifier f_S, both trained and evaluated on the training set S, and the true error $\epsilon(f_S)$ of that same classifier on the underlying population. This might look similar to the evaluation problem that we just addressed but there is a major difference. Earlier, the

classifier f was fixed and we only needed a dataset for evaluative purposes. And indeed, any fixed classifier does generalize: its error on a (previously unseen) dataset is an unbiased estimate of the population error. But what can we say when a classifier is trained and evaluated on the same dataset? Can we ever be confident that the training error will be close to the testing error?

Suppose that our learned classifier $f_{\mathcal{S}}$ must be chosen from some pre-specified set of functions $\mathcal{F}$. Recall from our discussion of test sets that while it is easy to estimate the error of a single classifier, things get hairy when we begin to consider collections of classifiers. Even if the empirical error of any one (fixed) classifier will be close to its true error with high probability, once we consider a collection of classifiers, we need to worry about the possibility that *just one* of them will receive a badly estimated error. The worry is that we might pick such a classifier and thereby grossly underestimate the population error. Moreover, even for linear models, because their parameters are continuously valued, we are typically choosing from an infinite class of functions ($|\mathcal{F}| = \infty$).

One ambitious solution to the problem is to develop analytic tools for proving uniform convergence, i.e., that with high probability, the empirical error rate for every classifier in the class $f \in \mathcal{F}$ will *simultaneously* converge to its true error rate. In other words, we seek a theoretical principle that would allow us to state that with probability at least $1 - \delta$ (for some small δ) no classifier's error rate $\epsilon(f)$ (among all classifiers in the class $\mathcal{F}$) will be misestimated by more than some small amount α. Clearly, we cannot make such statements for all model classes $\mathcal{F}$. Recall the class of memorization machines that always achieve empirical error 0 but never outperform random guessing on the underlying population.

In a sense the class of memorizers is too flexible. No such a uniform convergence result could possibly hold. On the other hand, a fixed classifier is useless—it generalizes perfectly, but fits neither the training data nor the test data. The central question of learning has thus historically been framed as a trade-off between more flexible (higher variance) model classes that better fit the training data but risk overfitting, versus more rigid (higher bias) model classes that generalize well but risk underfitting. A central question in learning theory has been to develop the appropriate mathematical analysis to quantify where a model sits along this spectrum, and to provide the associated guarantees.

In a series of seminal papers, Vapnik and Chervonenkis extended the theory on the convergence of relative frequencies to more general classes of functions (Vapnik and Chervonenkis, 1964, Vapnik and Chervonenkis, 1968, Vapnik and Chervonenkis, 1971, Vapnik and Chervonenkis, 1981, Vapnik and Chervonenkis, 1991, Vapnik and Chervonenkis, 1974). One of the key contributions of this line of work is the Vapnik–Chervonenkis (VC) dimension, which measures (one notion of) the complexity (flexibility) of a model class. Moreover, one of their key results bounds the difference between the empirical error and the population error as a function of the VC dimension and the number of samples:

$$P\left(R[p, f] - R_{\text{emp}}[\mathbf{X}, \mathbf{Y}, f] < \alpha\right) \geq 1 - \delta \text{ for } \alpha \geq c\sqrt{(\text{VC} - \log \delta)/n}. \tag{4.6.4}$$

Here $\delta > 0$ is the probability that the bound is violated, α is the upper bound on the generalization gap, and n is the dataset size. Lastly, $c > 0$ is a constant that depends only

on the scale of the loss that can be incurred. One use of the bound might be to plug in desired values of δ and α to determine how many samples to collect. The VC dimension quantifies the largest number of data points for which we can assign any arbitrary (binary) labeling and for each find some model f in the class that agrees with that labeling. For example, linear models on d-dimensional inputs have VC dimension $d + 1$. It is easy to see that a line can assign any possible labeling to three points in two dimensions, but not to four. Unfortunately, the theory tends to be overly pessimistic for more complex models and obtaining this guarantee typically requires far more examples than are actually needed to achieve the desired error rate. Note also that fixing the model class and δ, our error rate again decays with the usual $O(1/\sqrt{n})$ rate. It seems unlikely that we could do better in terms of n. However, as we vary the model class, VC dimension can present a pessimistic picture of the generalization gap.

4.6.4 Summary

The most straightforward way to evaluate a model is to consult a test set comprised of previously unseen data. Test set evaluations provide an unbiased estimate of the true error and converge at the desired $O(1/\sqrt{n})$ rate as the test set grows. We can provide approximate confidence intervals based on exact asymptotic distributions or valid finite sample confidence intervals based on (more conservative) finite sample guarantees. Indeed test set evaluation is the bedrock of modern machine learning research. However, test sets are seldom true test sets (used by multiple researchers again and again). Once the same test set is used to evaluate multiple models, controlling for false discovery can be difficult. This can cause huge problems in theory. In practice, the significance of the problem depends on the size of the holdout sets in question and whether they are merely being used to choose hyperparameters or if they are leaking information more directly. Nevertheless, it is good practice to curate real test sets (or multiple) and to be as conservative as possible about how often they are used.

Hoping to provide a more satisfying solution, statistical learning theorists have developed methods for guaranteeing uniform convergence over a model class. If indeed every model's empirical error simultaneously converges to its true error, then we are free to choose the model that performs best, minimizing the training error, knowing that it too will perform similarly well on the holdout data. Crucially, any one of such results must depend on some property of the model class. Vladimir Vapnik and Alexey Chernovenkis introduced the VC dimension, presenting uniform convergence results that hold for all models in a VC class. The training errors for all models in the class are (simultaneously) guaranteed to be close to their true errors, and guaranteed to grow even closer at $O(1/\sqrt{n})$ rates. Following the revolutionary discovery of VC dimension, numerous alternative complexity measures have been proposed, each facilitating an analogous generalization guarantee. See Boucheron *et al.* (2005) for a detailed discussion of several advanced ways of measuring function complexity. Unfortunately, while these complexity measures have become broadly useful tools in statistical theory, they turn out to be powerless (as straightforwardly applied) for explaining why deep neural networks generalize. Deep neural networks often have millions of parameters (or more), and can easily assign random labels to large collections of points. Nevertheless, they generalize well on practical problems and, surprisingly, they often gen-

eralize better, when they are larger and deeper, despite incurring higher VC dimensions. In the next chapter, we will revisit generalization in the context of deep learning.

4.6.5 Exercises

1. If we wish to estimate the error of a fixed model f to within 0.0001 with probability greater than 99.9%, how many samples do we need?
2. Suppose that somebody else possesses a labeled test set $\mathcal{D}$ and only makes available the unlabeled inputs (features). Now suppose that you can only access the test set labels by running a model f (with no restrictions placed on the model class) on each of the unlabeled inputs and receiving the corresponding error $\epsilon_{\mathcal{D}}(f)$. How many models would you need to evaluate before you leak the entire test set and thus could appear to have error 0, regardless of your true error?
3. What is the VC dimension of the class of fifth-order polynomials?
4. What is the VC dimension of axis-aligned rectangles on two-dimensional data?

[100]

Discussions[100].

4.7 Environment and Distribution Shift

In the previous sections, we worked through a number of hands-on applications of machine learning, fitting models to a variety of datasets. And yet, we never stopped to contemplate either where data came from in the first place or what we ultimately plan to do with the outputs from our models. Too often, machine learning developers in possession of data rush to develop models without pausing to consider these fundamental issues.

Many failed machine learning deployments can be traced back to this failure. Sometimes models appear to perform marvelously as measured by test set accuracy but fail catastrophically in deployment when the distribution of data suddenly shifts. More insidiously, sometimes the very deployment of a model can be the catalyst that perturbs the data distribution. Say, for example, that we trained a model to predict who will repay rather than default on a loan, finding that an applicant's choice of footwear was associated with the risk of default (Oxfords indicate repayment, sneakers indicate default). We might be inclined thereafter to grant a loan to any applicant wearing Oxfords and to deny all applicants wearing sneakers.

In this case, our ill-considered leap from pattern recognition to decision-making and our failure to critically consider the environment might have disastrous consequences. For starters, as soon as we began making decisions based on footwear, customers would catch on and change their behavior. Before long, all applicants would be wearing Oxfords, without any coincident improvement in credit-worthiness. Take a minute to digest this because

similar issues abound in many applications of machine learning: by introducing our model-based decisions to the environment, we might break the model.

While we cannot possibly give these topics a complete treatment in one section, we aim here to expose some common concerns, and to stimulate the critical thinking required to detect such situations early, mitigate damage, and use machine learning responsibly. Some of the solutions are simple (ask for the "right" data), some are technically difficult (implement a reinforcement learning system), and others require that we step outside the realm of statistical prediction altogether and grapple with difficult philosophical questions concerning the ethical application of algorithms.

4.7.1 Types of Distribution Shift

To begin, we stick with the passive prediction setting considering the various ways that data distributions might shift and what might be done to salvage model performance. In one classic setup, we assume that our training data was sampled from some distribution $p_S(\mathbf{x}, y)$ but that our test data will consist of unlabeled examples drawn from some different distribution $p_T(\mathbf{x}, y)$. Already, we must confront a sobering reality. Absent any assumptions on how p_S and p_T relate to each other, learning a robust classifier is impossible.

Consider a binary classification problem, where we wish to distinguish between dogs and cats. If the distribution can shift in arbitrary ways, then our setup permits the pathological case in which the distribution over inputs remains constant: $p_S(\mathbf{x}) = p_T(\mathbf{x})$, but the labels are all flipped: $p_S(y \mid \mathbf{x}) = 1 - p_T(y \mid \mathbf{x})$. In other words, if God can suddenly decide that in the future all "cats" are now dogs and what we previously called "dogs" are now cats—without any change in the distribution of inputs $p(\mathbf{x})$, then we cannot possibly distinguish this setting from one in which the distribution did not change at all.

Fortunately, under some restricted assumptions on the ways our data might change in the future, principled algorithms can detect shift and sometimes even adapt on the fly, improving on the accuracy of the original classifier.

Covariate Shift

Among categories of distribution shift, covariate shift may be the most widely studied. Here, we assume that while the distribution of inputs may change over time, the labeling function, i.e., the conditional distribution $P(y \mid \mathbf{x})$ does not change. Statisticians call this *covariate shift* because the problem arises due to a shift in the distribution of the covariates (features). While we can sometimes reason about distribution shift without invoking causality, we note that covariate shift is the natural assumption to invoke in settings where we believe that $\mathbf{x}$ causes y.

Consider the challenge of distinguishing cats and dogs. Our training data might consist of images of the kind in Fig. 4.7.1.

At test time we are asked to classify the images in Fig. 4.7.2.

The training set consists of photos, while the test set contains only cartoons. Training on a

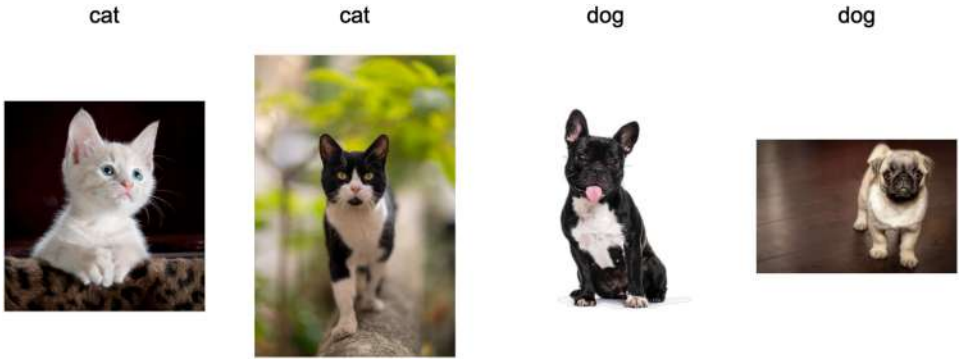


Fig. 4.7.1 Training data for distinguishing cats and dogs (illustrations: Lafeez Hossain / 500px / Getty Images; ilkermetinkursova / iStock / Getty Images Plus; GlobalP / iStock / Getty Images Plus; Musthafa Aboobakuru / 500px / Getty Images).

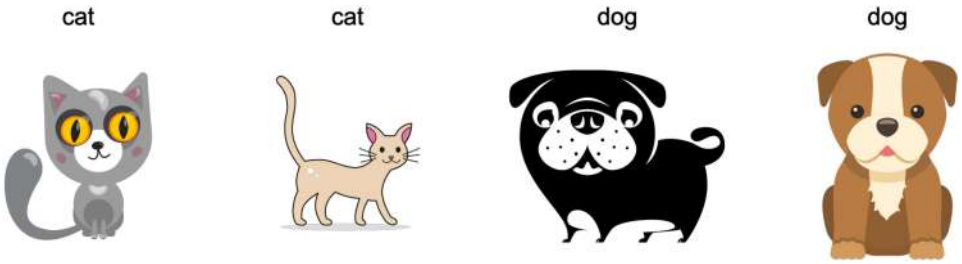


Fig. 4.7.2 Test data for distinguishing cats and dogs (illustrations: SIBAS_minich / iStock / Getty Images Plus; Ghrzuzudu / iStock / Getty Images Plus; id-work / DigitalVision Vectors / Getty Images; Yime / iStock / Getty Images Plus).

dataset with substantially different characteristics from the test set can spell trouble absent a coherent plan for how to adapt to the new domain.

Label Shift

Label shift describes the converse problem. Here, we assume that the label marginal $P(y)$ can change but the class-conditional distribution $P(\mathbf{x} \mid y)$ remains fixed across domains. Label shift is a reasonable assumption to make when we believe that y causes $\mathbf{x}$. For example, we may want to predict diagnoses given their symptoms (or other manifestations), even as the relative prevalence of diagnoses are changing over time. Label shift is the appropriate assumption here because diseases cause symptoms. In some degenerate cases the label shift and covariate shift assumptions can hold simultaneously. For example, when the label is deterministic, the covariate shift assumption will be satisfied, even when y causes $\mathbf{x}$. Interestingly, in these cases, it is often advantageous to work with methods that flow from the label shift assumption. That is because these methods tend to involve manipulating objects that look like labels (often low-dimensional), as opposed to objects that look like inputs, which tend to be high-dimensional in deep learning.

Concept Shift

We may also encounter the related problem of *concept shift*, which arises when the very definitions of labels can change. This sounds weird—a *cat* is a *cat*, no? However, other categories are subject to changes in usage over time. Diagnostic criteria for mental illness, what passes for fashionable, and job titles, are all subject to considerable amounts of concept shift. It turns out that if we navigate around the United States, shifting the source of our data by geography, we will find considerable concept shift regarding the distribution of names for *soft drinks* as shown in Fig. 4.7.3.

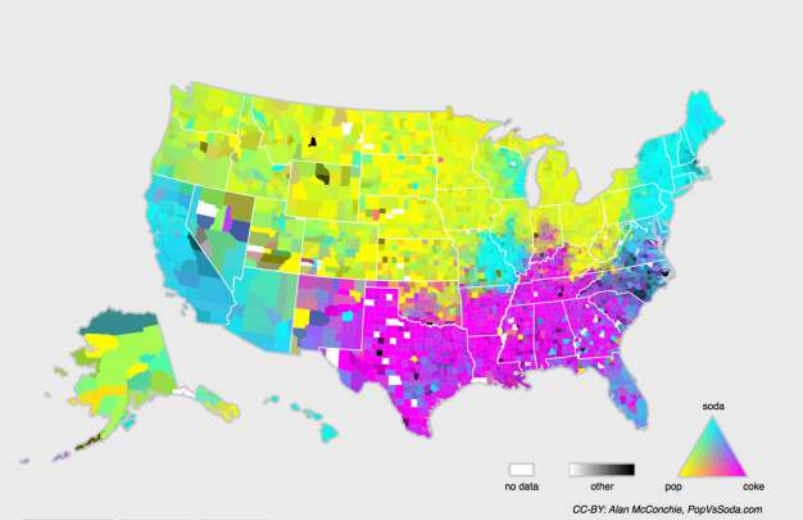


Fig. 4.7.3 Concept shift for soft drink names in the United States (CC-BY: Alan McConchie, PopVsSoda.com).

If we were to build a machine translation system, the distribution $P(y \mid \mathbf{x})$ might be different depending on our location. This problem can be tricky to spot. We might hope to exploit knowledge that shift only takes place gradually either in a temporal or geographic sense.

4.7.2 Examples of Distribution Shift

Before delving into formalism and algorithms, we can discuss some concrete situations where covariate or concept shift might not be obvious.

Medical Diagnostics

Imagine that you want to design an algorithm to detect cancer. You collect data from healthy and sick people and you train your algorithm. It works fine, giving you high accuracy and you conclude that you are ready for a successful career in medical diagnostics. *Not so fast.*

The distributions that gave rise to the training data and those you will encounter in the wild

might differ considerably. This happened to an unfortunate startup that some of we authors worked with years ago. They were developing a blood test for a disease that predominantly affects older men and hoped to study it using blood samples that they had collected from patients. However, it is considerably more difficult to obtain blood samples from healthy men than from sick patients already in the system. To compensate, the startup solicited blood donations from students on a university campus to serve as healthy controls in developing their test. Then they asked whether we could help them to build a classifier for detecting the disease.

As we explained to them, it would indeed be easy to distinguish between the healthy and sick cohorts with near-perfect accuracy. However, that is because the test subjects differed in age, hormone levels, physical activity, diet, alcohol consumption, and many more factors unrelated to the disease. This was unlikely to be the case with real patients. Due to their sampling procedure, we could expect to encounter extreme covariate shift. Moreover, this case was unlikely to be correctable via conventional methods. In short, they wasted a significant sum of money.

Self-Driving Cars

Say a company wanted to leverage machine learning for developing self-driving cars. One key component here is a roadside detector. Since real annotated data is expensive to get, they had the (smart and questionable) idea to use synthetic data from a game rendering engine as additional training data. This worked really well on "test data" drawn from the rendering engine. Alas, inside a real car it was a disaster. As it turned out, the roadside had been rendered with a very simplistic texture. More importantly, *all* the roadside had been rendered with the *same* texture and the roadside detector learned about this "feature" very quickly.

A similar thing happened to the US Army when they first tried to detect tanks in the forest. They took aerial photographs of the forest without tanks, then drove the tanks into the forest and took another set of pictures. The classifier appeared to work *perfectly*. Unfortunately, it had merely learned how to distinguish trees with shadows from trees without shadows—the first set of pictures was taken in the early morning, the second set at noon.

Nonstationary Distributions

A much more subtle situation arises when the distribution changes slowly (also known as *nonstationary distribution*) and the model is not updated adequately. Below are some typical cases.

- We train a computational advertising model and then fail to update it frequently (e.g., we forget to incorporate that an obscure new device called an iPad was just launched).
- We build a spam filter. It works well at detecting all spam that we have seen so far. But then the spammers wise up and craft new messages that look unlike anything we have seen before.

- We build a product recommendation system. It works throughout the winter but then continues to recommend Santa hats long after Christmas.

More Anecdotes

- We build a face detector. It works well on all benchmarks. Unfortunately it fails on test data—the offending examples are close-ups where the face fills the entire image (no such data was in the training set).
- We build a web search engine for the US market and want to deploy it in the UK.
- We train an image classifier by compiling a large dataset where each among a large set of classes is equally represented in the dataset, say 1000 categories, represented by 1000 images each. Then we deploy the system in the real world, where the actual label distribution of photographs is decidedly non-uniform.

4.7.3 Correction of Distribution Shift

As we have discussed, there are many cases where training and test distributions $P(\mathbf{x}, y)$ are different. In some cases, we get lucky and the models work despite covariate, label, or concept shift. In other cases, we can do better by employing principled strategies to cope with the shift. The remainder of this section grows considerably more technical. The impatient reader could continue on to the next section as this material is not prerequisite to subsequent concepts.

Empirical Risk and Risk

Let's first reflect on what exactly is happening during model training: we iterate over features and associated labels of training data $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and update the parameters of a model f after every minibatch. For simplicity we do not consider regularization, so we largely minimize the loss on the training:

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(f(\mathbf{x}_i), y_i), \tag{4.7.1}$$

where l is the loss function measuring "how bad" the prediction $f(\mathbf{x}_i)$ is given the associated label y_i. Statisticians call the term in $(4.7.1)$ *empirical risk*. The *empirical risk* is an average loss over the training data for approximating the *risk*, which is the expectation of the loss over the entire population of data drawn from their true distribution $p(\mathbf{x}, y)$:

$$E_{p(\mathbf{x},y)}[l(f(\mathbf{x}), y)] = \int \int l(f(\mathbf{x}), y) p(\mathbf{x}, y) \, d\mathbf{x} dy. \tag{4.7.2}$$

However, in practice we typically cannot obtain the entire population of data. Thus, *empirical risk minimization*, which is minimizing the empirical risk in $(4.7.1)$, is a practical strategy for machine learning, with the hope of approximately minimizing the risk.