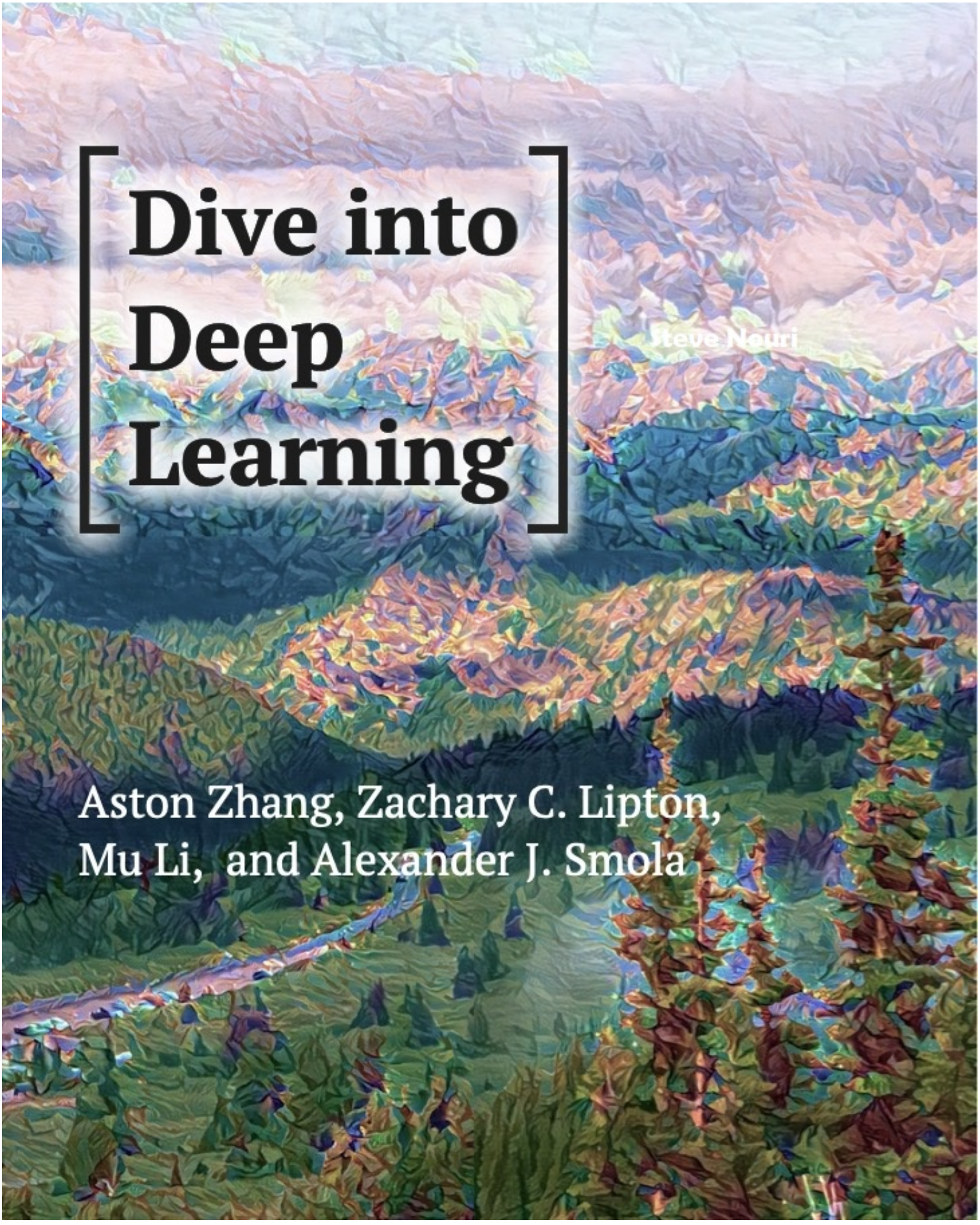

Dive into Deep Learning
Aston Zhang, Zachary C. Lipton,
Mu Li, and Alexander J. Smola

Dive into Deep Learning

Release 0.16.6

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola

Jun 25, 2021

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.

In just the past five years, deep learning has taken the world by surprise, driving rapid progress in such diverse fields as diverse as computer vision, natural language processing, automatic speech recognition, reinforcement learning, biomedical informatics, and 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 (and less autonomy than some companies might have you believe), smart reply systems that automatically draft the most mundane emails, helping people dig out from oppressively large inboxes, and software agents that dominate the world's best humans at board games like Go, a feat once thought to be decades away. Already, these tools exert ever-wider impacts on industry and society, changing the way movies are made, diseases are diagnosed, and playing a growing role in basic sciences—from astrophysics to biology.

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 needing 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. Take dynamic web applications as an example. Despite a large number of companies, like Amazon, developing successful database-driven web applications in the 1990s, the potential of this technology to aid creative en-

trepreneurs has been realized to a far greater degree 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 basic statistical principles and intuitions that help us to extract generalizable insights from data; and (v) the engineering required to train models efficiently, navigating the pitfalls of numerical computing and getting the most out of available hardware. Teaching both 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) were up to date; (ii) covered the full breadth of modern machine learning with substantial technical depth; and (iii) interleaved exposition of the quality one expects from an engaging textbook with the clean runnable code that one expects to find in hands-on tutorials. We found plenty of code examples for 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, ResNets, 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[3], 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 of the concepts behind 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.

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[4] for interactive discussion of technical details and to answer questions.

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. At present there exist no tools and no workflow perfectly suited to these demands, so we had to assemble our own. We describe our approach in detail in Section 19.6. 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 to generate multiple outputs, and Discourse for the forum. While our system is not yet perfect, these choices provide a good compromise among the competing concerns. We believe that this might be the first book published using such an integrated workflow.

[3] http://distill.pub
[4] http://discuss.d2l.ai

Learning by Doing

Many textbooks present concepts in succession, covering each in exhaustive detail. For example, Chris Bishop's excellent textbook (Bishop, 2006), teaches each topic so thoroughly that getting to the chapter on linear regression requires a non-trivial 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 will 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 probability distributions.

Aside from a few preliminary notebooks that provide a crash course in the basic mathematical background, each subsequent chapter introduces both a reasonable number of new concepts and provides single self-contained working examples—using real datasets. This presents 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. On the other hand, 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.

We will interleave the runnable code with background material as needed. In general, we will often err on the side of making tools available before explaining them fully (and we will follow up by explaining the background later). For instance, we might use *stochastic gradient descent* before fully explaining why it is useful or 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 will teach deep learning concepts from scratch. Sometimes, we want to delve into fine details about the models that would typically be hidden from the user by deep learning frameworks' advanced abstractions. 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 will often present two versions of the example: one where we implement everything from scratch, relying only on NumPy-like functionality and automatic differentiation, and another, more practical example, where we write succinct code using the high-level APIs of deep learning frameworks. Once we have taught you how some component works, we can just use the high-level APIs in subsequent tutorials.

Content and Structure

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

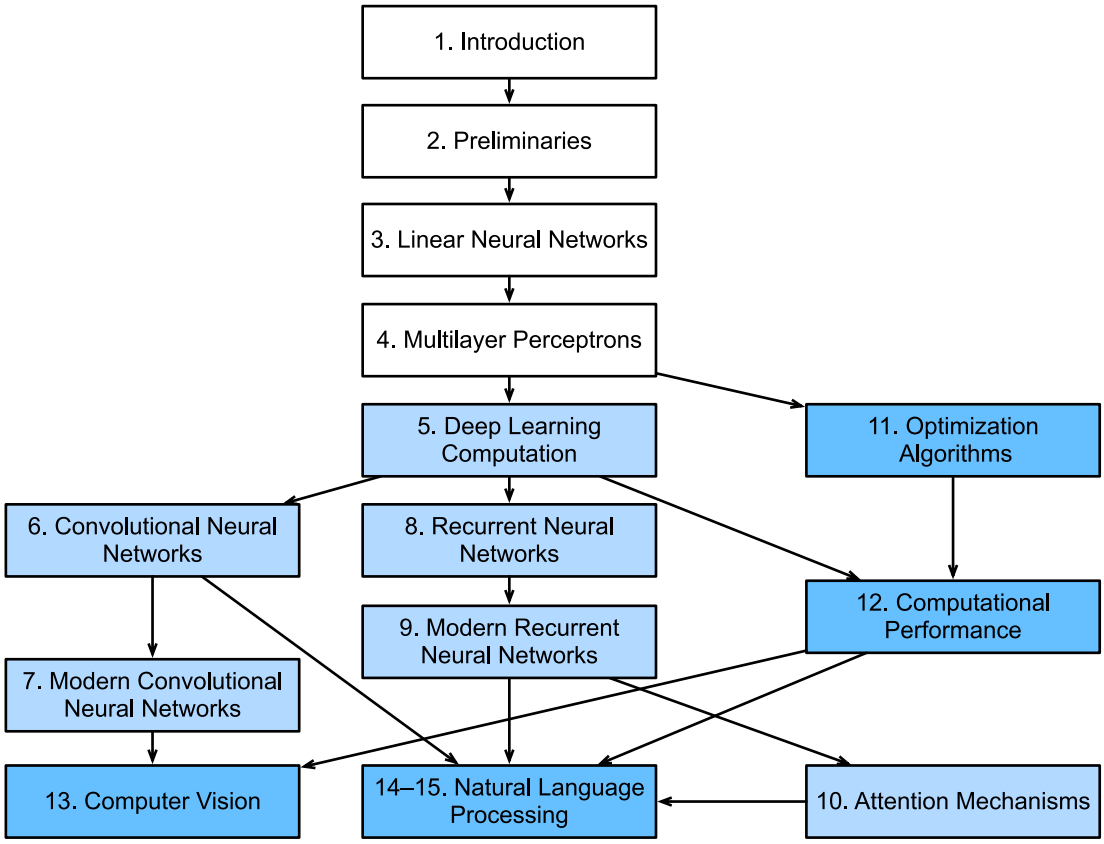


Fig. 1: Book structure

- The first part covers basics and preliminaries. Chapter 1 offers 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 basic concepts from linear algebra, calculus, and probability. Chapter 3 and Chapter 4 cover the most basic concepts and techniques in deep learning, including regression and classification; linear models and multilayer perceptrons; and overfitting and regularization.
- The next five chapters focus on modern deep learning techniques. Chapter 5 describes the key computational components of deep learning systems and lays the groundwork for our subsequent implementations of more complex models. Next, Chapter 6 and Chapter 7, introduce convolutional neural networks (CNNs), powerful tools that form the backbone of most modern computer vision systems. Similarly, Chapter 8 and Chapter 9 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 10, we introduce 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.
- Part three discusses scalability, efficiency, and applications. First, in Chapter 11, we dis-

cuss several common optimization algorithms used to train deep learning models. The next chapter, Chapter 12, examines several key factors that influence the computational performance of your deep learning code. In Chapter 13, we illustrate major applications of deep learning in computer vision. In Chapter 14 and Chapter 15, we show 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, today deep learning practitioners today must often tread where no cogent theory can provide firm guidance. Despite our best attempts, formal explanations for the efficacy of various techniques are still lacking, both because the mathematics to characterize these models can be so difficult and also because serious inquiry on these topics has only just recently kicked into high gear. We are hopeful that as the theory of deep learning progresses, future editions of this book can provide insights that eclipse those presently available.

To avoid unnecessary repetition, we encapsulate some of our most frequently imported and referred-to functions and classes in the `d2l` package. To indicate a block of code, such as a function, class, or collection of import statements, that will be subsequently accessed via the `d2l` package, we will mark it with `#@save`. We offer a detailed overview of these functions and classes in Section 19.7. The `d2l` package is lightweight and only requires the following dependencies:

```
#@save
import collections
import hashlib
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

d2l = sys.modules[__name__]
```

Most of the code in this book is based on Apache MXNet, an open-source framework for deep learning that is the preferred choice of AWS (Amazon Web Services), as well as many colleges and companies. All of the code in this book has passed tests under the newest MXNet version. However, due to the rapid development of deep learning, some code *in the print edition* may not work properly in future versions of MXNet. We plan to keep the online version up-to-date. In case you encounter any problems, please consult *Installation* (page 9) to update your code and runtime environment.

Here is how we import modules from MXNet.

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

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 come in with some basics, including modest amounts of linear algebra, calculus, probability, and Python programming. Just in case you for get the basics, the Appendix provides a refresher on most of the mathematics you will find in this book. Most of the time, 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 Bela Bollobas (Bollobas, 1999) covers linear algebra and functional analysis in great depth. All of Statistics (Wasserman, 2013) provides a marvelous introduction to statistics. Joe Blitzsteins books and courses[5] on probability and inference are pegagogical gems. And if you have not used Python before, you may want to peruse this Python tutorial[6].

Forum

Associated with this book, we have launched a discussion forum, located at discuss.d2l.ai[7]. When 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. Specifically, 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

[5] https://projects.iq.harvard.edu/stat110/home
[6] http://learnpython.org/
[7] https://discuss.d2l.ai/

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, Steven-Jokes, 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.

We thank Amazon Web Services, especially Swami Sivasubramanian, Raju Gulabani, Charlie Bell, 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.

Summary

- Deep learning has revolutionized pattern recognition, introducing technology that now powers a wide range of technologies, including computer vision, natural language processing, automatic speech recognition.
- To successfully apply deep learning, you must understand how to cast a problem, the 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.
- To answer questions related to this book, visit our forum at https://discuss.d2l.ai/.
- All notebooks are available for download on GitHub.

Exercises

1. Register an account on the discussion forum of this book discuss.d2l.ai[8].
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.

[8] https://discuss.d2l.ai/

Discussions[9]

[9] https://discuss.d2l.ai/t/18

Installation

In order to get you up and running for hands-on learning experience, we need to set you up with an environment for running Python, Jupyter notebooks, the relevant libraries, and the code needed to run the book itself.

Installing Miniconda

The simplest way to get going will be to install Miniconda[10]. The Python 3.x version is required. You can skip the following steps if conda has already been installed.

Visit the Miniconda website and determine the appropriate version for your system based on your Python 3.x version and machine architecture. For example, if you are using macOS and Python 3.x you would download the bash script with strings "Miniconda3" and "MacOSX" in its name, navigate to the download location and execute the installation as follows:

```
sh Miniconda3-latest-MacOSX-x86_64.sh -b
```

A Linux user with Python 3.x would download the file with strings "Miniconda3" and "Linux" in its name and execute the following at the download location:

```
sh Miniconda3-latest-Linux-x86_64.sh -b
```

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

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

Now close and re-open your current shell. You should be able to create a new environment as following:

```
conda create --name d2l python=3.8 -y
```

[10] https://conda.io/en/latest/miniconda.html

Downloading the D2L Notebooks

Next, we need to download the code of this book. You can click the "All Notebooks" tab on the top of any HTML page to download and unzip the code. Alternatively, if you have `unzip` (otherwise run `sudo apt install unzip`) available:

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

Now we will want to activate the `d2l` environment.

```
conda activate d2l
```

Installing the Framework and the `d2l` Package

Before installing the 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 do not count for our purposes). If you are installing on a GPU server, proceed to *GPU Support* (page 11) for instructions to install a GPU-supported version.

Otherwise, you can install the CPU version as follows. That will be more than enough horsepower to get you through the first few chapters but you will want to access GPUs before running larger models.

```
pip install mxnet==1.7.0.post1
```

We also install the `d2l` package that encapsulates frequently used functions and classes in this book.

```
# -U: Upgrade all packages to the newest available version
pip install -U d2l
```

Once they are installed, we now open the Jupyter notebook by running:

```
jupyter notebook
```

At this point, you can open http://localhost:8888 (it usually opens automatically) in your Web browser. Then we can run the code for each section of the book. Please always execute `conda activate d2l` to activate the runtime environment before running the code of the book or updating the deep learning framework or the `d2l` package. To exit the environment, run `conda deactivate`.

GPU Support

By default, MXNet is installed without GPU support to ensure that it will run on any computer (including most laptops). Part of this book requires or recommends running with GPU. If your computer has NVIDIA graphics cards and has installed CUDA[11], then you should install a GPU-enabled version. If you have installed the CPU-only version, you may need to remove it first by running:

```
pip uninstall mxnet
```

Then we need to find the CUDA version you installed. You may check it through `nvcc --version` or `cat /usr/local/cuda/version.txt`. Assume that you have installed CUDA 10.1, then you can install with the following command:

```
# For Windows users
pip install mxnet-cu101==1.7.0 -f https://dist.mxnet.io/python

# For Linux and macOS users
pip install mxnet-cu101==1.7.0
```

You may change the last digits according to your CUDA version, e.g., `cu100` for CUDA 10.0 and `cu90` for CUDA 9.0.

Exercises

1. Download the code for the book and install the runtime environment.

Discussions[12]

[11] https://developer.nvidia.com/cuda-downloads
[12] https://discuss.d2l.ai/t/23

Notation

The notation used throughout this book is summarized below.

Numbers

- x: A scalar
- $\mathbf{x}$: A vector
- $\mathbf{X}$: A matrix
- X: A tensor
- $\mathbf{I}$: An identity matrix
- 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 $\mathcal{B}$
- $\mathcal{A} \setminus \mathcal{B}$: Subtraction of set $\mathcal{B}$ from set $\mathcal{A}$

Functions and Operators

- $f(\cdot)$: A function
- $\log(\cdot)$: The natural logarithm
- $\exp(\cdot)$: The exponential function
- $\mathbf{1}_{\mathcal{X}}$: The indicator function
- $(\cdot)^\top$: Transpose of a vector or a matrix
- $\mathbf{X}^{-1}$: Inverse of matrix $\mathbf{X}$
- $\odot$: Hadamard (elementwise) product
- $[\cdot, \cdot]$: Concatenation
- $|\mathcal{X}|$: Cardinality of set $\mathcal{X}$
- $\|\cdot\|_p$: L_p norm
- $\|\cdot\|$: L_2 norm
- $\langle \mathbf{x}, \mathbf{y} \rangle$: Dot product of vectors $\mathbf{x}$ and $\mathbf{y}$
- $\sum$: Series addition
- $\prod$: Series multiplication
- $\stackrel{\text{def}}{=}$: Definition

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

- $P(\cdot)$: Probability distribution
- $z \sim P$: Random variable z has probability distribution P
- $P(X \mid Y)$: Conditional probability of $X \mid Y$
- $p(x)$: Probability density function
- $E_x[f(x)]$: Expectation of f with respect to 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 random variable Z
- $\mathrm{Var}(X)$: Variance of random variable X
- σ_X: Standard deviation of random variable X
- $\mathrm{Cov}(X, Y)$: Covariance of random variables X and Y
- $\rho(X, Y)$: Correlation of random variables X and Y
- $H(X)$: Entropy of random variable X
- $D_{\mathrm{KL}}(P\|Q)$: KL-divergence of distributions P and Q

Complexity

- $\mathcal{O}$: Big O notation

Discussions[13]

[13] https://discuss.d2l.ai/t/25

1 Introduction

Until recently, nearly every computer program that we interact with daily was coded by software developers from first principles. 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 would come up with the broad strokes of a working solution that might probably look something like this: (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 in methodical detail the appropriate action that our program should take in every conceivable circumstance.

To build the brains of our application, we would have to step through every possible corner case that we anticipate encountering, devising appropriate rules. Each time a customer clicks to add an item to their shopping cart, we add an entry to the shopping cart database table, associating that user's ID with the requested product's ID. 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 could write such a program from first principles and confidently launch it *before* ever seeing a real customer. Our ability to design automated systems from first principles 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 should not be using 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 question, expressed in free-form text, and answers it correctly.
- Write a program that given an image can identify all the people it contains, drawing 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.

In each of these cases, even elite programmers are incapable of coding up solutions from scratch. The reasons for this can vary. Sometimes the program that we are looking for follows a pattern that changes over time, and we need our programs to adapt. In other cases, the relationship (say between pixels, and abstract categories) may be too complicated, requiring thousands or millions of computations that are beyond our conscious understanding even if our eyes manage the task

effortlessly. *Machine learning* is the study of powerful techniques 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 performs according to 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, and focus 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 we fabricated this story 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 44000 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 $\{\text{yes}, \text{no}\}$ on 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.

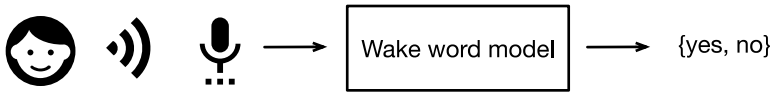


Fig. 1.1.1: Identify a wake word.

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 and label those that do and that do not contain the wake word. In the machine learning approach, 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 set of parameters, those that improve the performance of our program with respect to some measure of performance on the task of interest.

You can think of the parameters as knobs that we can turn, manipulating the behavior of the program. Fixing the parameters, 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 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 $\{\text{yes}, \text{no}\}$ labels).
3. Tweak the knobs so the model sucks less with respect to those examples.
4. Repeat Step 2 and 3 until the model is awesome.

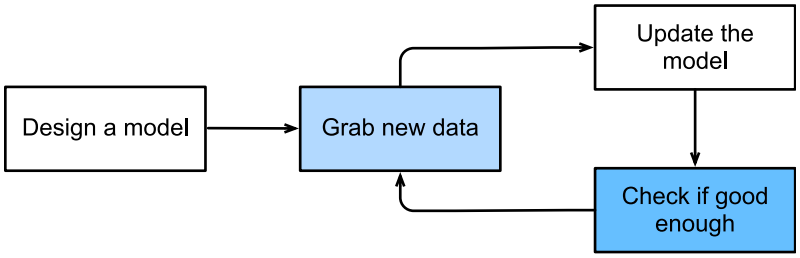


Fig. 1.1.2: A typical training process.

To summarize, rather than code up a wake word recognizer, we code up a program that can *learn* to recognize wake words, if we present it 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, and 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. Later we will take a deep dive into different machine learning problems. First, 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 take on:

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 constitutes data, but for now, we will err on the practical side and focus on the key properties to 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* (or *covariates*), from which the model must make its predictions. In the supervised learning problems above, the thing to predict is a special attribute that is designated as the *label* (or *target*).

If we were working with image data, each individual photograph might constitute an example, each represented by an ordered list of numerical values corresponding to the brightness of each pixel. A 200×200 color photograph would consist of $200 \times 200 \times 3 = 120000$ numerical values, corresponding to the brightness of the red, green, and blue channels for each spatial location. In another traditional task, we might try to predict whether or not a patient will survive, given a standard set of features such as age, vital signs, and diagnoses.

When every example is characterized by the same number of numerical values, we say that the data consist of fixed-length vectors and we describe the constant length of the vectors as the *dimensionality* of the data. As you might imagine, fixed-length can be a convenient property. If we wanted to train a model to recognize cancer in microscopy images, fixed-length inputs mean we have one less thing 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 to all show up with the same resolution or shape. For images, we might consider cropping them all to a standard size, but that strategy only gets us so far. We risk losing information in the cropped out portions. Moreover, text data resist 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 pre-conceived 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 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 are 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 common failure mode occurs in datasets where some groups of people are unrepresented in the training data. Imagine applying a skin cancer recognition system in the wild that had never seen black skin before. Failure can also occur when the data do not merely under-represent some groups but reflect 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 to update our model, and some people might disagree on whether the proposed update constituted an improvement or a decline.

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 lower is 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. 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. Others (e.g., error rate) are difficult to optimize directly, owing to non-differentiability or other complications. In these cases, it is common to optimize a *surrogate objective*.

Typically, the loss function is defined with respect to the model's parameters and depends upon the dataset. 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 fitting model parameters) and the *test dataset* (or *test set*, which is held out for evaluation), reporting how the model performs on both of them. You could think of training performance as being like a student's scores on 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. In other words, the test performance can deviate significantly from the training performance. When a model performs well on the training set but fails to generalize to unseen data, we say that it is *overfitting*. In real-life terms, this is like flunking the real exam despite doing well on practice exams.

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 short, at each step, this method checks to see, for each parameter, which way the training set loss would move if you perturbed that parameter just a small amount. It then updates the parameter in the direction that may reduce the loss.

1.3 Kinds of Machine Learning Problems

The wake word problem in our motivating example is just one among many problems that machine learning can tackle. To motivate the reader further and provide us with some common language when we talk about more problems throughout the book, in the following we list a sampling of machine learning problems. We will constantly refer to our aforementioned concepts such as data, models, and training techniques.

1.3.1 Supervised Learning

Supervised learning addresses the task of predicting labels 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. Our goal is to produce a model that maps any input to a label prediction.

To ground this description in a concrete example, if we were working in healthcare, then we might want to predict whether or not a patient would have a heart attack. This observation, "heart attack" or "no heart attack", would be our label. The input features might be vital signs such as heart rate, diastolic blood pressure, and systolic blood pressure.

The supervision comes into play because for choosing the parameters, we (the supervisors) provide the model with a dataset consisting of labeled examples, where each example is matched with

the ground-truth label. 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 within machine learning, 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.

Even with the simple description "predicting labels given input features" supervised learning can take a great many forms and require a great many modeling decisions, depending on (among other considerations) the type, size, and the number of inputs and outputs. For example, we use different models to process sequences of arbitrary lengths and for processing 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.

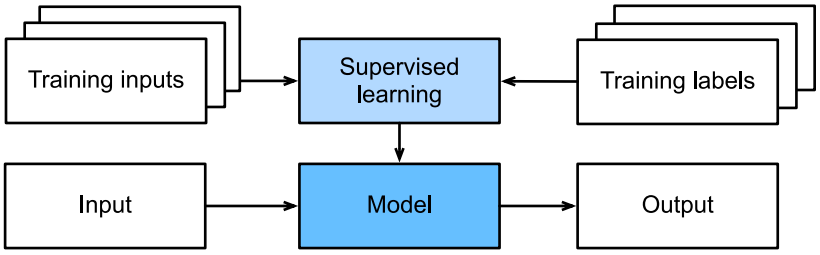


Fig. 1.3.1: Supervised learning.

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, where 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 $[3000, 4, 3, 10]$. Feature vectors like this are essential for most classic machine learning algorithms.

What makes a problem a regression is actually the output. 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 like above. The label, the price of sale, is a numerical value. When labels take on arbitrary numerical values, we call this a *regression* problem. Our goal is to produce a model whose predictions closely approximate the actual label values.

Lots of practical problems are well-described 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[14]. Predicting the length of stay for patients in the hospital is also a regression problem. A good rule of thumb is that any *how much?* or *how many?* problem should suggest regression, such as:

- 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 he sent you a bill of 350 dollars. Now imagine that your friend hired the same contractor for 2 hours and that he 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 variance owes to a few factors besides 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 bend comfortably to this template. For example, a bank wants to add check scanning to its mobile app. This would involve the customer snapping a photo of a check with their smart phone's camera and the app would need to be able to automatically understand text seen in the image. Specifically, it would also need to understand handwritten text to be even more robust, such as mapping a handwritten character to one of the known characters. This kind of *which one?* problem is called *classification*. It is treated with a different set of algorithms than 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 which *category* (formally called *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 $\{\text{cat}, \text{dog}\}$. While in regression, we sought a regressor to output a numerical value, in classification, we seek a classifier, whose output is the predicted class assignment.

[14] https://en.wikipedia.org/wiki/Netflix_Prize

For reasons that we will get into as the book gets more technical, it can be hard to optimize a model that can only output a hard categorical assignment, e.g., either "cat" or "dog". In these cases, it is usually much easier to instead 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 $\{\text{cat}, \text{dog}\}$, 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 one notion of uncertainty. It is not the only notion of uncertainty and we will discuss others in more advanced chapters.

When we have more than two possible classes, we call the problem *multiclass classification*. Common examples include hand-written character recognition $\{0, 1, 2, ...9, \text{a}, \text{b}, \text{c}, ...\}$. 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 can be demystified via an introduction to information theory in subsequent 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.

Fig. 1.3.2: Death cap—do not eat!

Now, assume that you built a classifier and trained it to predict if a mushroom is poisonous based on a photograph. Say our poison-detection classifier outputs that the probability that Fig. 1.3.2 contains 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, we need to compute the expected risk that we incur as the loss function, i.e., we need to multiply the probability of the outcome with the benefit (or harm) associated with it. In this case, the loss incurred by eating the mushroom can 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 actually is a death cap.

Classification can get much more complicated than just binary, multiclass, or even multi-label classification. For instance, there are some variants of classification for addressing hierarchies. Hierarchies assume that there exist some relationships among the many classes. So not all errors are equal—if we must err, we would prefer to misclassify to a related class rather than to a distant class. Usually, this is referred to as *hierarchical classification*. One early example is due to Linnaeus[15], who organized the animals in a hierarchy.

[15] https://en.wikipedia.org/wiki/Carl_Linnaeus

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

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 in Fig. 1.3.3.

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

As you can see, there is a cat in Fig. 1.3.3, and a rooster, a dog, and a donkey, with some trees in the background. Depending on what we want to do with our model ultimately, treating this as a binary classification problem might not make a lot of sense. Instead, we might want to give the model the option of saying the image depicts a cat, a dog, a donkey, *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 as multi-label classification problems. 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 because these concepts are correlated. Posts about "cloud

computing" are likely to mention "AWS" and posts about "machine learning" could also deal with "programming languages".

We also have to deal with this kind of problem when dealing with the biomedical literature, where correctly tagging articles is important because it allows researchers to do exhaustive reviews of the literature. At the National Library of Medicine, a number of professional annotators go over each article that gets indexed in PubMed to associate it with the relevant terms from MeSH, a collection of roughly 28000 tags. This is a time-consuming process and the annotators typically have a one-year lag between archiving and tagging. Machine learning can be used here to provide provisional tags until each article can have a proper manual review. Indeed, for several years, the BioASQ organization has hosted competitions[16] to do precisely this.

Search

Sometimes we do not just want to assign each example to a bucket or to a real value. In the field of information retrieval, we want to impose a ranking on a set of items. Take web search for an example. The goal is less to determine whether a particular page is relevant for a query, but rather, which one of the plethora of search results is most relevant for a particular user. We really care about the ordering of the relevant search results and our learning algorithm needs to produce ordered subsets of elements from a larger set. In other words, if we are asked to produce the first 5 letters from the alphabet, there is a difference between returning "A B C D E" and "C A B E D". Even if the result set is the same, the ordering within the set matters.

One possible solution to this problem is to first assign to every element in the set a corresponding relevance score and then to retrieve the top-rated elements. PageRank[17], the original secret sauce behind the Google search engine was an early example of such a scoring system but it was peculiar in that it did not depend on the actual query. Here they relied on a simple relevance filter to identify the set of relevant items and then on PageRank to order those results that contained the query term. Nowadays, search engines use machine learning and behavioral models to obtain query-dependent relevance scores. There are entire academic conferences devoted to this subject.

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 relevant items 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, and GoodReads). In some other cases, they provide implicit feedback, e.g., by skipping titles on a playlist, which might indicate dissatisfaction but might just indicate that the song was inappropriate in context. In the simplest formulations, these systems are trained to estimate some score, such as an estimated rating or the probability of purchase, given a user and an item.

[16] http://bioasq.org/
[17] https://en.wikipedia.org/wiki/PageRank

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 is an example of deep learning books recommended by Amazon based on personalization algorithms tuned to capture one's preferences.

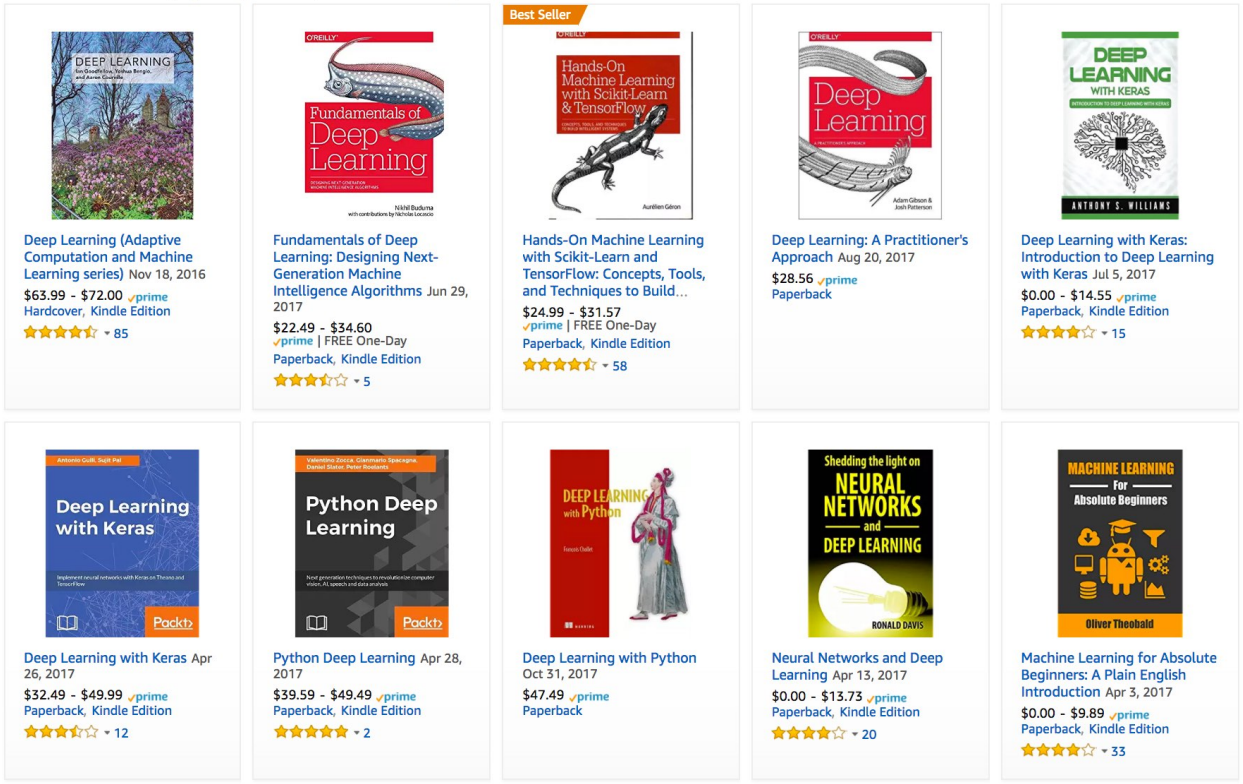


Fig. 1.3.4: Deep learning books recommended by Amazon.

Despite their tremendous economic value, recommendation 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 five and one 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 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 from a fixed set of features: square footage, number of bedrooms, number of bathrooms, walking time to downtown. We also discussed mapping from an image (of fixed dimension) to the predicted probabilities that it belongs to each of a fixed number of classes, or taking a user ID and a product ID, and predicting a star rating. In these cases, once we feed our fixed-length input into the model to generate an output, the model immediately forgets what it just saw.

This might be fine if our inputs truly all have the same dimensions and if successive inputs truly have nothing to do with each other. But how would 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. Same goes for language. One popular deep learning problem is machine translation: the task of ingesting sentences in some source language and predicting their translation in another language.

These problems also occur in medicine. We might want a model to monitor patients in the intensive care unit and to fire off alerts if their risk of death in the next 24 hours exceeds some threshold. We definitely would not want this model to throw away everything it knows about the patient history each hour and just make its predictions based on the most recent measurements.

These problems are among the most exciting applications of machine learning and they are instances of *sequence learning*. They require a model to either ingest sequences of inputs or to emit sequences of outputs (or both). Specifically, *sequence to sequence learning* considers problems where input and output are both variable-length sequences, such as machine translation and transcribing text from the spoken speech. 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. In other words, the number of inputs and outputs is essentially the same. For instance, we might want to know where the verbs and subjects are. Alternatively, we might want to know which words are the named entities. In general, the goal is to decompose and annotate text based on structural and grammatical assumptions to get some annotation. This sounds more complex than it actually is. Below is a very simple example of annotating a sentence with tags indicating which words refer to named entities (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 (shown in Fig. 1.3.5), and the output is the textual 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.

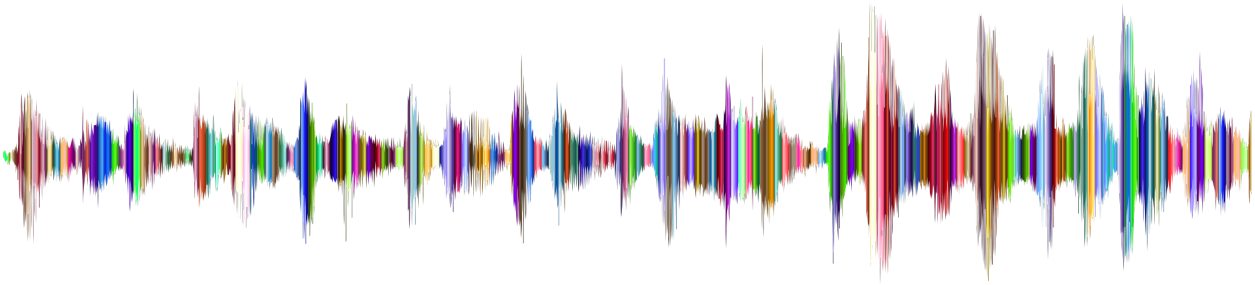


Fig. 1.3.5: `-D-e-e-p- L-ea-r-ni-ng-` in an audio recording.

Text to Speech. This is the inverse of automatic speech recognition. In other words, the input is text and the output is an audio file. In this case, the output is much longer than the input. While it is easy for humans to recognize a bad audio file, this is not quite so trivial for computers.

Machine Translation. Unlike the case of speech recognition, where corresponding inputs and outputs occur in the same order (after alignment), in machine translation, order inversion can be vital. In other words, while we are still converting one sequence into another, neither the number of inputs and outputs nor the order of corresponding data examples are assumed to be the same. 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:          Did you already check out this excellent tutorial?
Wrong alignment:  Did you yourself already this excellent tutorial 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. These are active areas of research.

1.3.2 Unsupervised learning

All the examples so far were related to supervised learning, i.e., situations 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 banal boss. The boss stands over your shoulder and tells you exactly what to do in every situation until you learn to map from situations to actions. Working for such a boss sounds pretty lame. On the other hand, it is easy to please this boss. You just recognize the pattern as quickly as possible and imitate their actions.

In a completely opposite way, 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. We call this class of problems *unsupervised learning*, and the type and number of questions we could 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 quite 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* address this problem.
- Another important and exciting recent development in unsupervised learning is the advent of *generative adversarial networks*. These give us a procedural way to synthesize data, even complicated structured data like images and audio. The underlying statistical mechanisms are tests to check whether real and fake data are the same.

1.3.3 Interacting with an Environment

So far, we have not discussed where data actually come 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 either 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 of the learning takes place after the algorithm is disconnected from the environment, this is sometimes called *offline learning*. For supervised learning, the process by considering data collection from an environment looks like Fig. 1.3.6.

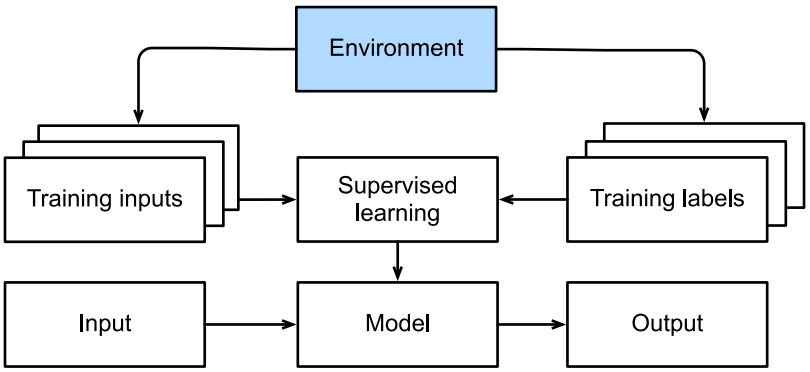


Fig. 1.3.6: Collecting data for supervised learning from an environment.

This simplicity of offline learning has its charms. The upside is that we can worry about pattern recognition in isolation, without any distraction from these other problems. But the downside is that the problem formulation is quite limiting. If you are more ambitious, or if you grew up reading Asimov's Robot series, then you might imagine artificially intelligent bots 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. Moreover, unlike predictions, actions actually impact the environment. If we want to train an intelligent agent, we must account for the way its actions might impact the future observations of the agent.

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, i.e., an adversarial setting like spam filtering (against spammers) or playing a game (vs. an opponent)?
- Does the environment not care?
- Does the environment have shifting dynamics? For example, does future data always resemble the past or do the patterns change over time, either naturally or in response to our automated tools?

This last question raises the problem of *distribution shift*, when training and test data are different. It is a problem that most of us have experienced when taking exams written by a lecturer, while the homework was composed by his teaching assistants. Next, we will briefly describe reinforcement learning, a setting that explicitly considers interactions 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, and the AlphaGo program that dethroned the world champion at the board game Go 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). Finally, 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 an reinforcement learning agent is governed by a policy. In short, a *policy* is just a function that maps from observations of the environment to actions. The goal of reinforcement learning is to produce a good policy.

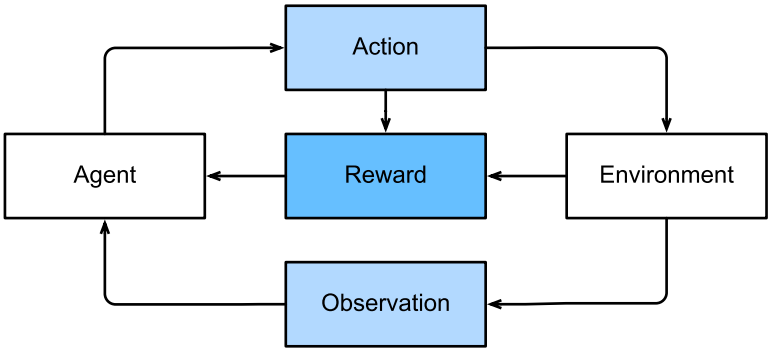


Fig. 1.3.7: The interaction between reinforcement learning and an environment.

It is hard to overstate the generality of the reinforcement learning framework. For example, we can cast any supervised learning problem as a reinforcement learning problem. 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.

That being said, 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 for example the game of chess. The only real reward signal comes at the end of the game when we either win, which we might assign a reward of 1, or when we lose, which we could assign a reward of -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 large number of well-chosen actions over the previous year. Getting more promotions in the future requires figuring out what actions along the way led to the promotion.

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 a cleaning robot found itself trapped in one of many identical closets in a house. Inferring the precise location (and thus state) of the robot might require considering its previous observations before 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-run reward in exchange for knowledge.

The general reinforcement learning problem is a very general setting. Actions affect subsequent observations. Rewards are only observed corresponding to the chosen actions. The environment may be either fully or partially observed. Accounting for all this complexity at once may ask too much of researchers. 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 the problem a *contextual bandit problem*. When there is no state, just a set of available actions with initially unknown rewards, this problem is 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 solving them. Although many deep learning methods are recent inventions, the core idea of programming with data and neural networks (names of many deep learning models) has been studied for centuries. In fact, humans have held the desire to analyze data and to predict future outcomes for long and much of natural science has its roots in this. For instance, the Bernoulli distribution is named after Jacob Bernoulli (1655–1705)[18], and the Gaussian distribution was discovered by Carl Friedrich Gauss (1777–1855)[19]. He invented, for instance, the least mean squares algorithm, which is still used today for countless problems from insurance calculations to medical diagnostics. These tools gave rise to an 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)[20] illustrates averaging the length of 16 adult men's feet to obtain the average foot length.

Fig. 1.4.1: Estimating the length of a foot.

Fig. 1.4.1 illustrates how this estimator works. The 16 adult men were asked to line up in a row, when leaving the church. Their aggregate length was then divided by 16 to obtain an estimate for what now amounts to 1 foot. This "algorithm" was later improved to deal with misshapen feet—the 2 men with the shortest and longest feet respectively were sent away, averaging only over the remainder. This is one of the earliest examples of the trimmed mean estimate.

[18] https://en.wikipedia.org/wiki/Jacob_Bernoulli
[19] https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss
[20] https://www.maa.org/press/periodicals/convergence/mathematical-treasures-jacob-kobels-geometry

Statistics really took off with the collection and availability of data. One of its titans, Ronald Fisher (1890–1962)[21], contributed significantly to its theory and also its applications in genetics. Many of his algorithms (such as linear discriminant analysis) and formula (such as the Fisher information matrix) are still in frequent use today. In fact, even the Iris dataset that Fisher released in 1936 is still used sometimes to illustrate machine learning algorithms. He was also a proponent of eugenics, which should remind us that the morally dubious use of data science has as long and enduring a history as its productive use in industry and the natural sciences.

A second influence for machine learning came from information theory by Claude Shannon (1916–2001)[22] and the theory of computation via Alan Turing (1912–1954)[23]. Turing posed the question "can machines think?" in his famous paper *Computing Machinery and Intelligence* (Turing, 1950). In what he described as the Turing test, a machine can be considered *intelligent* if it is difficult for a human evaluator to distinguish between the replies from a machine and a human based on textual interactions.

Another influence can be found in neuroscience and psychology. After all, humans clearly exhibit intelligent behavior. It is thus only reasonable to ask whether one could explain and possibly reverse engineer this capacity. One of the oldest algorithms inspired in this fashion was formulated by Donald Hebb (1904–1985)[24]. In his groundbreaking book *The Organization of Behavior* (Hebb & Hebb, 1949), he posited that neurons learn by positive reinforcement. This became known as the Hebbian learning rule. It is the prototype of Rosenblatt's perceptron learning algorithm and it 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.

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 1932 was a popular tool for testing the efficacy of algorithms. The MNIST dataset with its 60000 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. Unlike neural networks, they did not require weeks to train and provided predictable results with strong theoretical guarantees.

[21] https://en.wikipedia.org/wiki/Ronald_Fisher
[22] https://en.wikipedia.org/wiki/Claude_Shannon
[23] https://en.wikipedia.org/wiki/Alan_Turing
[24] https://en.wikipedia.org/wiki/Donald_O._Hebb

1.5 The Road to Deep Learning

Much of this changed with the ready availability of large amounts of data, due to the World Wide Web, the advent of companies serving hundreds of millions of users online, a dissemination of cheap, high-quality sensors, cheap data storage (Kryder's law), and cheap computation (Moore's law), in particular in the form of GPUs, originally engineered for computer gaming. Suddenly algorithms and models that seemed computationally infeasible became relevant (and vice versa). This is best illustrated in Table 1.5.1.

Table 1.5.1: Dataset vs. computer memory and computational power

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)

It is evident that random-access memory has not kept pace with the growth in data. At the same time, the increase in computational power has outpaced that of the data available. This means that statistical models need to become more memory efficient (this is typically achieved by adding nonlinearities) while simultaneously being able to spend more time on optimizing these parameters, due to an increased computational 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 & Pitts, 1943), convolutional neural networks (LeCun et al., 1998), long short-term memory (Hochreiter & Schmidhuber, 1997), and Q-Learning (Watkins & Dayan, 1992), were essentially "rediscovered" in the past decade, after laying 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 below barely scratches the surface of the ideas that have 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 the danger of overfitting. This was achieved by applying noise injection (Bishop, 1995) throughout the neural network, replacing weights by random variables for training purposes.
- 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 sequence.

- Multi-stage designs, e.g., via the memory networks (Sukhbaatar et al., 2015) and the neural programmer-interpreter (Reed & DeFreitas, 2015) allowed 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, similar to how a processor can modify memory for a computation.
- Another key development 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, it opened up density estimation to a wide variety of techniques. Examples of galloping Zebras (Zhu et al., 2017) and of fake celebrity faces (Karras et al., 2017) are both testimony to this progress. Even amateur doodlers can produce photorealistic images based on just sketches that describe how the layout of a scene looks like (Park et al., 2019).
- In many cases, a single GPU is insufficient to process 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 1024 GPUs with a minibatch size of, say 32 images per batch amounts to an aggregate minibatch of about 32000 images. Recent work, first by Li (Li, 2017), and subsequently by (You et al., 2017) and (Jia et al., 2018) pushed the size up to 64000 observations, reducing training time for the ResNet-50 model on the ImageNet dataset to less than 7 minutes. For comparison—initially training times were measured in the order of days.
- The ability to parallelize computation has also contributed quite crucially to progress in reinforcement learning, at least whenever simulation is an option. This has led to significant progress in computers achieving superhuman performance in Go, Atari games, Starcraft, and in physics simulations (e.g., using MuJoCo). See e.g., (Silver et al., 2016) for a description of how to achieve this in AlphaGo. In a nutshell, reinforcement learning works best if plenty of (state, action, reward) triples are available, i.e., whenever it is possible to try out lots of things to learn how they relate to each other. Simulation provides such an avenue.
- Deep learning frameworks have played a crucial role in disseminating ideas. The first generation of frameworks allowing for easy modeling encompassed Caffe[25], Torch[26], and Theano[27]. Many seminal papers were written using these tools. By now, they have been superseded by TensorFlow[28] (often used via its high level API Keras[29]), CNTK[30], Caffe 2[31], and Apache MXNet[32]. The third generation of tools, namely imperative tools for deep learning, was arguably spearheaded by Chainer[33], which used a syntax similar to Python NumPy to

[25] https://github.com/BVLC/caffe
[26] https://github.com/torch
[27] https://github.com/Theano/Theano
[28] https://github.com/tensorflow/tensorflow
[29] https://github.com/keras-team/keras
[30] https://github.com/Microsoft/CNTK
[31] https://github.com/caffe2/caffe2
[32] https://github.com/apache/incubator-mxnet
[33] https://github.com/chainer/chainer

describe models. This idea was adopted by both PyTorch[34], the Gluon API[35] of MXNet, and Jax[36].

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 less than 10 lines of code, putting it firmly into the grasp of programmers.

1.6 Success Stories

AI has a long history of delivering results that would be difficult to accomplish otherwise. For instance, the 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 automatically. 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 answer spoken questions with a reasonable degree of accuracy. This includes menial tasks such as turning on light switches (a boon to the disabled) up to making 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 the ability to recognize speech accurately. Gradually the accuracy of such systems has increased to the point where they reach human parity for certain applications (Xiong et al., 2018).
- Object recognition likewise has come a long way. Estimating 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-5 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 birds or diagnosing skin cancer.
- Games used to be a bastion of human intelligence. 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. Unlike 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

[34] https://github.com/pytorch/pytorch
[35] https://github.com/apache/incubator-mxnet
[36] https://github.com/google/jax

large and it is not fully observed (we do not know the opponents' cards). Libratus exceeded human performance in Poker using efficiently structured strategies (Brown & Sandholm, 2017). This illustrates the impressive progress in games and the fact that advanced algorithms played a crucial part in them.

- Another indication of progress in AI is the advent of self-driving cars and trucks. While full autonomy is not quite within reach yet, excellent progress has been made in this direction, with companies such as Tesla, NVIDIA, and Waymo shipping products that enable at least 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 computer vision aspect of these problems. The rest is heavily tuned by engineers.

Again, the above list barely scratches the surface of where machine learning has impacted practical applications. 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. Machine learning is thus becoming a ubiquitous tool for engineers and scientists.

Frequently, the question of the AI apocalypse, or the AI singularity has been raised in non-technical articles on AI. The fear is that somehow machine learning systems will become sentient and decide independently from their programmers (and masters) about things that directly affect the livelihood of humans. To some extent, AI already affects the livelihood of humans in an immediate way: 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 is ready to manipulate its human creators (or burn their coffee). 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 tools for *artificial general intelligence* simply do not exist 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 menial tasks fulfilled by truck drivers and shop assistants can and will be automated. Farm robots will likely reduce the cost for organic farming but they will also automate harvesting operations. This phase of the industrial revolution may have profound consequences on large swaths of society, since truck drivers and shop assistants are some of the most common jobs 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 a much more pressing concern than the potential of malevolent superintelligence to destroy humanity.

1.7 Characteristics

Thus far, we have talked about machine learning broadly, which is both a branch of AI and an approach to AI. Though deep learning is a subset of machine learning, the dizzying set of algorithms and applications makes it difficult to assess what specifically the ingredients for deep learning might be. This is as difficult as trying to pin down required ingredients for pizza since almost every component is substitutable.

As we have described, machine learning can use data to learn transformations between inputs and outputs, such as transforming audio into text in speech recognition. In doing so, it is often necessary to represent data in a way suitable for algorithms to transform such representations into the output. *Deep learning* is *deep* in precisely the sense that its models learn many *layers* of transformations, where each layer offers the representation at one level. For example, layers near the input may represent low-level details of the data, while layers closer to the classification output may represent more abstract concepts used for discrimination. Since *representation learning* aims at finding the representation itself, deep learning can be referred to as multi-level representation learning.

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 where traditional machine learning 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 the use of *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 little 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, yielding 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, 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 are scarce, one needs to rely on simplifying assumptions about reality in order to obtain useful models. When data are abundant, this can be replaced by nonparametric models that fit reality more accurately. 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 parametric approximations of how electrons behave by hand, 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 explainability.

Another difference to previous work is the acceptance of suboptimal solutions, dealing with non-

convex nonlinear optimization problems, and the willingness to try things before proving them. This newfound empiricism in dealing with statistical problems, combined with a rapid influx of talent has led to rapid progress 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 everyone to learn about deep learning and we hope that our readers will benefit from this.

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. Deep learning is 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.
- Whole system optimization is a key component in obtaining high performance. The availability of efficient deep learning frameworks has made design and implementation of this significantly easier.

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?
2. Which problems that you encounter have many examples for how to solve them, yet no specific way to automate them? These may be prime candidates for using deep learning.
3. Viewing the development of AI as a new industrial revolution, what is the relationship between algorithms and data? Is it similar to steam engines and coal? What is the fundamental difference?
4. Where else can you apply the end-to-end training approach, such as in Fig. 1.1.2, physics, engineering, and econometrics?

Discussions[37]

[37] https://discuss.d2l.ai/t/22

2 | Preliminaries

To get started with deep learning, we will need to develop a few basic skills. All machine learning is concerned with extracting information from data. So we will begin by learning the practical skills for storing, manipulating, and preprocessing data.

Moreover, machine learning typically requires working with large datasets, which we can think of as tables, where the rows correspond to examples and the columns correspond to attributes. Linear algebra gives us a powerful set of techniques for working with tabular data. We will not go too far into the weeds but rather focus on the basic of matrix operations and their implementation.

Additionally, deep learning is all about optimization. We have a model with some parameters and we want to find those that fit our data *the best*. Determining which way to move each parameter at each step of an algorithm requires a little bit of calculus, which will be briefly introduced. Fortunately, the `autograd` package automatically computes differentiation for us, and we will cover it next.

Next, machine learning is concerned with making predictions: what is the likely value of some unknown attribute, given the information that we observe? To reason rigorously under uncertainty we will need to invoke the language of probability.

In the end, the official documentation provides plenty of descriptions and examples that are beyond this book. To conclude the chapter, we will show you how to look up documentation for the needed information.

This book has kept the mathematical content to the minimum necessary to get a proper understanding of deep learning. However, it does not mean that this book is mathematics free. Thus, this chapter provides a rapid introduction to basic and frequently-used mathematics to allow anyone to understand at least *most* of the mathematical content of the book. If you wish to understand *all* of the mathematical content, further reviewing the online appendix on mathematics[38] should be sufficient.

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 let us get our hands dirty first by playing with synthetic data. To start, we introduce the n-dimensional array, which is also called the *tensor*.

If you have worked with NumPy, the most widely-used scientific computing package in Python, then you will find this section familiar. No matter which framework you use, its *tensor class*

[38] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/index.html

(ndarray in MXNet, Tensor in both PyTorch and TensorFlow) is similar to NumPy's ndarray with a few killer features. First, GPU is well-supported to accelerate the computation whereas NumPy only supports CPU computation. Second, the tensor class supports automatic differentiation. These properties make the tensor class suitable for deep learning. Throughout the book, when we say tensors, we are referring to instances of the tensor class unless otherwise stated.

2.1.1 Getting Started

In this section, we aim to get you up and running, equipping you with the basic math and numerical computing tools that you will build on as you progress through the book. Do not worry if you struggle to grok some of the mathematical concepts or library functions. The following sections will revisit this material in the context of practical examples and it will sink in. On the other hand, if you already have some background and want to go deeper into the mathematical content, just skip this section.

To start, we import the np (numpy) and npx (numpy_extension) modules from MXNet. Here, the np module includes functions supported by NumPy, while the npx module contains a set of extensions developed to empower deep learning within a NumPy-like environment. When using tensors, we almost always invoke the set_np function: this is for compatibility of tensor processing by other components of MXNet.

```
from mxnet import np, npx

npx.set_np()
```

A tensor represents a (possibly multi-dimensional) array of numerical values. With one axis, a tensor corresponds (in math) to a *vector*. With two axes, a tensor corresponds to a *matrix*. Tensors with more than two axes do not have special mathematical names.

To start, we can use arange to create a row vector x containing the first 12 integers starting with 0, though they are created as floats by default. Each of the values in a tensor is called an *element* of the tensor. For instance, there are 12 elements in the tensor x. Unless otherwise specified, a new tensor will be stored in main memory and designated for CPU-based computation.

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

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

We can access a tensor's *shape* (the length along each axis) by inspecting its shape property.

```
x.shape
```

```
(12,)
```

If we just want to know the total number of elements in a tensor, i.e., the product of all of the shape elements, we can inspect its size. Because we are dealing with a vector here, the single element of its shape is identical to its size.

```
x.size
```

```
12
```

To change the shape of a tensor without altering either the number of elements or their values, we can invoke the `reshape` function. For example, we can transform our tensor, `x`, from a row vector with shape (12,) to a matrix with shape (3, 4). This new tensor contains the exact same values, but views them as a matrix organized as 3 rows and 4 columns. To reiterate, although the shape has changed, the elements have not. Note that the size is unaltered by reshaping.

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

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

Reshaping by manually specifying every dimension is unnecessary. If our target shape is a matrix with shape (height, width), then after we know the width, the height is given implicitly. Why should we have to perform the division ourselves? In the example above, to get a matrix with 3 rows, we specified both that it should have 3 rows and 4 columns. Fortunately, tensors can automatically work out one dimension given the rest. We invoke this capability by placing `-1` for the dimension that we would like tensors to automatically infer. In our case, instead of calling `x.reshape(3, 4)`, we could have equivalently called `x.reshape(-1, 4)` or `x.reshape(3, -1)`.

Typically, we will want our matrices initialized either with zeros, ones, some other constants, or numbers randomly sampled from a specific distribution. We can create a tensor representing a tensor with all elements set to 0 and a shape of (2, 3, 4) as follows:

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

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

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

Similarly, we can create tensors with each element set to 1 as follows:

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

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

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

Often, we want to randomly sample the values for each element in a tensor from some probability distribution. For example, when we construct arrays to serve as parameters in a neural network,

we will typically initialize their values randomly. The following snippet creates a tensor with shape (3, 4). Each of its elements is randomly sampled from a standard Gaussian (normal) distribution with a mean of 0 and a standard deviation of 1.

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

```
array([[ 2.2122064 ,  1.1630787 ,  0.7740038 ,  0.4838046 ],
       [ 1.0434405 ,  0.29956347,  1.1839255 ,  0.15302546],
       [ 1.8917114 , -1.1688148 , -1.2347414 ,  1.5580711 ]])
```

We can also specify the exact values for each element in the desired tensor by supplying a Python list (or list of lists) containing the numerical values. Here, the outermost list corresponds to axis 0, and the inner list to axis 1.

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

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

2.1.2 Operations

This book is not about software engineering. Our interests are not limited to simply reading and writing data from/to arrays. We want to perform mathematical operations on those arrays. Some of the simplest and most useful operations are the *elementwise* operations. These apply a standard scalar operation to each element of an array. For functions that take two arrays as inputs, elementwise operations apply some standard binary operator on each pair of corresponding elements from the two arrays. We can create an elementwise function from any function that maps from a scalar to a scalar.

In mathematical notation, we would denote such a *unary* scalar operator (taking one input) by the signature $f : \mathbb{R} \rightarrow \mathbb{R}$. This just means that the function is mapping from any real number ($\mathbb{R}$) onto another. Likewise, we denote a *binary* scalar operator (taking two real inputs, and yielding one output) by 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 (`+`, `-`, `*`, `/`, and `**`) have all been *lifted* to elementwise operations for any identically-shaped tensors of arbitrary shape. We can call elementwise operations on any two tensors of the same shape. In the following example, we use commas to formulate a 5-element tuple, where each element is the result of an elementwise operation.

Operations

The common standard arithmetic operators (`+`, `-`, `*`, `/`, and `**`) have all been *lifted* to elementwise operations.

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

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

Many more operations can be applied elementwise, including unary operators like exponentiation.

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

```
array([2.7182817e+00, 7.3890562e+00, 5.4598148e+01, 2.9809580e+03])
```

In addition to elementwise computations, we can also perform linear algebra operations, including vector dot products and matrix multiplication. We will explain the crucial bits of linear algebra (with no assumed prior knowledge) in Section 2.3.

We can also *concatenate* multiple tensors together, stacking them end-to-end to form a larger tensor. 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, the first element of the shape) vs. columns (axis 1, the second element of the shape). We can see that the first output tensor's axis-0 length (6) is the sum of the two input tensors' axis-0 lengths ($3 + 3$); while the second output tensor's axis-1 length (8) is the sum of the two input tensors' axis-1 lengths ($4 + 4$).

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

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

Sometimes, we want to construct a binary tensor via *logical statements*. Take `X == Y` as an example. For each position, if `X` and `Y` are equal at that position, the corresponding entry in the new tensor takes a value of 1, meaning that the logical statement `X == Y` is true at that position; otherwise that position takes 0.

```
X == Y
```

```
array([[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()
```

```
array(66.)
```

2.1.3 Broadcasting Mechanism

In the above section, we saw how to perform elementwise operations on two tensors of the same shape. Under certain conditions, even when shapes differ, we can still perform elementwise operations by invoking the *broadcasting mechanism*. This mechanism works in the following way: First, expand one or both arrays by copying elements appropriately so that after this transformation, the two tensors have the same shape. Second, carry out the elementwise operations on the resulting arrays.

In most cases, we broadcast along an axis where an array initially only has length 1, such as in the following example:

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

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

Since `a` and `b` are 3×1 and 1×2 matrices respectively, their shapes do not match up if we want to add them. We *broadcast* the entries of both matrices into a larger 3×2 matrix as follows: for matrix `a` it replicates the columns and for matrix `b` it replicates the rows before adding up both elementwise.

```
a + b
```

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

2.1.4 Indexing and Slicing

Just as in any other Python array, elements in a tensor can be accessed by index. As in any Python array, the first element has index 0 and ranges are specified to include the first but *before* the last element. As in standard Python lists, we can access elements according to their relative position to the end of the list by using negative indices.

Thus, `[-1]` selects the last element and `[1:3]` selects the second and the third elements as follows:

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

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

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

```
X[1, 2] = 9
X
```

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

If we want to assign multiple elements the same value, we simply index all of them and then assign them the value. For instance, `[0:2, :]` accesses the first and second rows, where `:` takes all the elements along axis 1 (column). While we discussed indexing for matrices, this obviously also works for vectors and for tensors of more than 2 dimensions.

```
X[0:2, :] = 12
X
```

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

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 will dereference the tensor that `Y` used to point to and instead point `Y` at the newly allocated memory. In the following example, we demonstrate this with Python's `id()` function, which gives us the exact address of the referenced object in memory. After running `Y = Y + X`, we will find that `id(Y)` points to a different location. That is because Python first evaluates `Y + X`, allocating new memory for the result and then makes `Y` point 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 might have hundreds of megabytes of parameters and update all of them multiple times per second. Typically, we will 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, other references will still point to the old memory location, making it possible for parts of our code to inadvertently reference stale parameters.

Fortunately, performing in-place operations is easy. We can assign the result of an operation to a previously allocated array with slice notation, e.g., `Y[:] = <expression>`. To illustrate this concept, we first create a new matrix `Z` with the same shape as another `Y`, using `zeros_like` to allocate a block of 0 entries.

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

```
id(Z): 140221399309824
id(Z): 140221399309824
```

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, or vice versa, is easy. The converted result does not share memory. This minor inconvenience is actually quite important: when you perform operations on the CPU or on GPUs, you do not want to halt computation, waiting to see whether the NumPy package of Python might want to be doing something else with the same chunk of memory.

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

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

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

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

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

Summary

- The main interface to store and manipulate data for deep learning is the tensor (n-dimensional array). It provides a variety of functionalities including basic mathematics operations, broadcasting, indexing, slicing, memory saving, and conversion to other Python objects.

Exercises

1. Run the code in this section. Change the conditional statement `X == Y` in this section 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?

Discussions[39]

2.2 Data Preprocessing

So far we have introduced a variety of techniques for manipulating data that are already stored in tensors. To apply deep learning to solving real-world problems, we often begin with preprocessing raw data, rather than those nicely prepared data in the tensor format. Among popular data analytic tools in Python, the `pandas` package is commonly used. Like many other extension packages in the vast ecosystem of Python, `pandas` can work together with tensors. So, we will briefly walk through steps for preprocessing raw data with `pandas` and converting them into the tensor format. We will cover more data preprocessing techniques in later chapters.

2.2.1 Reading the Dataset

As an example, we begin by creating an artificial dataset that is stored in a csv (comma-separated values) file `../data/house_tiny.csv`. Data stored in other formats may be processed in similar ways.

Below we write the dataset row by row into a csv file.

```
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,Alley,Price\n')  # Column names
    f.write('NA,Pave,127500\n')  # Each row represents a data example
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

[39] https://discuss.d2l.ai/t/26

To load the raw dataset from the created csv file, we import the `pandas` package and invoke the `read_csv` function. This dataset has four rows and three columns, where each row describes the number of rooms ("NumRooms"), the alley type ("Alley"), and the price ("Price") of a house.

```
# If pandas is not installed, just uncomment the following line:
# !pip install pandas
import pandas as pd

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

```
   NumRooms Alley   Price
0       NaN  Pave  127500
1       2.0   NaN  106000
2       4.0   NaN  178100
3       NaN   NaN  140000
```

2.2.2 Handling Missing Data

Note that "NaN" entries are missing values. To handle missing data, typical methods include *imputation* and *deletion*, where imputation replaces missing values with substituted ones, while deletion ignores missing values. Here we will consider imputation.

By integer-location based indexing (`iloc`), we split `data` into `inputs` and `outputs`, where the former takes the first two columns while the latter only keeps the last column. For numerical values in `inputs` that are missing, we replace the "NaN" entries with the mean value of the same column.

```
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

```
   NumRooms Alley
0       3.0  Pave
1       2.0   NaN
2       4.0   NaN
3       3.0   NaN
```

For categorical or discrete values in `inputs`, we consider "NaN" as a category. Since the "Alley" column only takes two types of categorical values "Pave" and "NaN", `pandas` can automatically convert this column to two columns "Alley_Pave" and "Alley_nan". A row whose alley type is "Pave" will set values of "Alley_Pave" and "Alley_nan" to 1 and 0. A row with a missing alley type will set their values to 0 and 1.

```
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

```
   NumRooms  Alley_Pave  Alley_nan
0       3.0           1          0
1       2.0           0          1
2       4.0           0          1
3       3.0           0          1
```

2.2.3 Conversion to the Tensor Format

Now that all the entries in `inputs` and `outputs` are numerical, they can be converted to the tensor format. Once data are in this format, they can be further manipulated with those tensor functionalities that we have introduced in Section 2.1.

```
from mxnet import np

X, y = np.array(inputs.values), np.array(outputs.values)
X, y
```

```
(array([[3., 1., 0.],
        [2., 0., 1.],
        [4., 0., 1.],
        [3., 0., 1.]], dtype=float64),
 array([127500, 106000, 178100, 140000], dtype=int64))
```

Summary

- Like many other extension packages in the vast ecosystem of Python, `pandas` can work together with tensors.
- Imputation and deletion can be used to handle missing data.

Exercises

Create a raw dataset with more rows and columns.

1. Delete the column with the most missing values.
2. Convert the preprocessed dataset to the tensor format.

Discussions[40]

2.3 Linear Algebra

Now that you can store and manipulate data, let us briefly review the subset of basic linear algebra that you will need to understand and implement most of models covered in this book. Below, we introduce the basic mathematical objects, arithmetic, and operations in linear algebra, expressing each of them through mathematical notation and the corresponding implementation in code.

[40] https://discuss.d2l.ai/t/28

2.3.1 Scalars

If you never studied linear algebra or machine learning, then your past experience with math probably consisted of thinking about one number at a time. And, if you ever balanced a checkbook or even paid for dinner at a restaurant then you already know how to do basic things like adding and multiplying pairs of numbers. For example, the temperature in Palo Alto is 52 degrees Fahrenheit. Formally, we call values consisting of just one numerical quantity *scalars*. If you wanted to convert this value to Celsius (the metric system's more sensible temperature scale), you would evaluate the expression $c = \frac{5}{9}(f - 32)$, setting f to 52. In this equation, each of the terms—5, 9, and 32—are scalar values. The placeholders c and f are called *variables* and they represent unknown scalar values.

In this book, we adopt the mathematical notation where scalar variables are denoted by ordinary lower-cased letters (e.g., x, y, and z). We denote the space of all (continuous) *real-valued* scalars by $\mathbb{R}$. For expedience, we will punt on rigorous definitions of what precisely *space* is, but just remember for now that the expression $x \in \mathbb{R}$ is a formal way to say that x is a real-valued scalar. The symbol $\in$ can be pronounced "in" and simply denotes membership in a set. Analogously, we could write $x, y \in \{0, 1\}$ to state that x and y are numbers whose value can only be 0 or 1.

A scalar is represented by a tensor with just one element. In the next snippet, we instantiate two scalars and perform some familiar arithmetic operations with them, namely addition, multiplication, division, and exponentiation.

```
from mxnet import np, npx

npx.set_np()

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

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

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

2.3.2 Vectors

You can think of a vector as simply a list of scalar values. We call these values the *elements* (*entries* or *components*) of the vector. When our vectors represent examples from our dataset, their values hold some real-world significance. For example, if we were training a model to predict the risk that a loan defaults, we might associate each applicant with a vector whose components correspond to their income, length of employment, number of previous defaults, and other factors. If we were studying the risk of heart attacks hospital patients potentially face, we might represent each patient by a vector whose components capture their most recent vital signs, cholesterol levels, minutes of exercise per day, etc. In math notation, we will usually denote vectors as bold-faced, lower-cased letters (e.g., $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$).

We work with vectors via one-dimensional tensors. In general tensors can have arbitrary lengths, subject to the memory limits of your machine.

```
x = np.arange(4)
x
```

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

We can refer to any element of a vector by using a subscript. For example, we can refer to the i^{th} element of $\mathbf{x}$ by x_i. Note that the element x_i is a scalar, so we do not bold-face the font when referring to it. Extensive literature considers column vectors to be the default orientation of vectors, so does this book. In math, a vector $\mathbf{x}$ can be written as

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{2.3.1}$$

where $x_1, \ldots, x_n$ are elements of the vector. In code, we access any element by indexing into the tensor.

```
x[3]
```

```
array(3.)
```

Length, Dimensionality, and Shape

Let us revisit some concepts from Section 2.1. A vector is just an array of numbers. And just as every array has a length, so does every vector. In math notation, if we want to say that a vector $\mathbf{x}$ consists of n real-valued scalars, we can express this as $\mathbf{x} \in \mathbb{R}^n$. The length of a vector is commonly called the *dimension* of the vector.

As with an ordinary Python array, we can access the length of a tensor by calling Python's built-in `len()` function.

```
len(x)
```

```
4
```

When a tensor represents a vector (with precisely one axis), we can also access its length via the `.shape` attribute. The shape is a tuple that lists the length (dimensionality) along each axis of the tensor. For tensors with just one axis, the shape has just one element.

```
x.shape
```

```
(4,)
```

Note that the word "dimension" tends to get overloaded in these contexts and this tends to confuse people. To clarify, we use the dimensionality of a *vector* or an *axis* to refer to its length, i.e., the number of elements of a vector or an axis. However, we use the dimensionality of a tensor to refer to the number of axes that a tensor has. In this sense, the dimensionality of some axis of a tensor will be the length of that axis.

2.3.3 Matrices

Just as vectors generalize scalars from order zero to order one, matrices generalize vectors from order one to order two. Matrices, which we will typically denote with bold-faced, capital letters (e.g., **X**, **Y**, and **Z**), are represented in code as tensors with two axes.

In math notation, we use $\mathbf{A} \in \mathbb{R}^{m \times n}$ to express that the matrix **A** consists of m rows and n columns of real-valued scalars. Visually, we can illustrate any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ as a table, where each element a_{ij} belongs to the i^{th} row and j^{th} column:

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

For any $\mathbf{A} \in \mathbb{R}^{m \times n}$, the shape of **A** is (m, n) or $m \times n$. Specifically, when a matrix has the same number of rows and columns, its shape becomes a square; thus, it is called a *square matrix*.

We can create an $m \times n$ matrix by specifying a shape with two components m and n when calling any of our favorite functions for instantiating a tensor.

```
A = np.arange(20).reshape(5, 4)
A
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.],
       [16., 17., 18., 19.]])
```

We can access the scalar element a_{ij} of a matrix **A** in (2.3.2) by specifying the indices for the row (i) and column (j), such as $[\mathbf{A}]_{ij}$. When the scalar elements of a matrix **A**, such as in (2.3.2), are not given, we may simply use the lower-case letter of the matrix **A** with the index subscript, a_{ij}, to refer to $[\mathbf{A}]_{ij}$. To keep notation simple, commas are inserted to separate indices only when necessary, such as $a_{2,3j}$ and $[\mathbf{A}]_{2i-1,3}$.

Sometimes, we want to flip the axes. When we exchange a matrix's rows and columns, the result is called the *transpose* of the matrix. Formally, we signify a matrix **A**'s transpose by $\mathbf{A}^\top$ and if $\mathbf{B} = \mathbf{A}^\top$, then $b_{ij} = a_{ji}$ for any i and j. Thus, the transpose of **A** in (2.3.2) is a $n \times m$ matrix:

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

Now we access a matrix's transpose in code.

```
A.T
```

```
array([[ 0.,  4.,  8., 12., 16.],
       [ 1.,  5.,  9., 13., 17.],
       [ 2.,  6., 10., 14., 18.],
       [ 3.,  7., 11., 15., 19.]])
```

As a special type of the square matrix, a *symmetric matrix* $\mathbf{A}$ is equal to its transpose: $\mathbf{A} = \mathbf{A}^\top$. Here we define a symmetric matrix `B`.

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

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

Now we compare `B` with its transpose.

```
B == B.T
```

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

Matrices are useful data structures: they allow us to organize data that have different modalities of variation. For example, rows in our matrix might correspond to different houses (data examples), while columns might correspond to different attributes. This should sound familiar if you have ever used spreadsheet software or have read Section 2.2. Thus, although the default orientation of a single vector is a column vector, in a matrix that represents a tabular dataset, it is more conventional to treat each data example as a row vector in the matrix. And, as we will see in later chapters, this convention will enable common deep learning practices. For example, along the outermost axis of a tensor, we can access or enumerate minibatches of data examples, or just data examples if no minibatch exists.

2.3.4 Tensors

Just as vectors generalize scalars, and matrices generalize vectors, we can build data structures with even more axes. Tensors ("tensors" in this subsection refer to algebraic objects) give us a generic way of describing n-dimensional arrays with an arbitrary number of axes. Vectors, for example, are first-order tensors, and matrices are second-order tensors. Tensors are denoted with capital letters of a special font face (e.g., `X`, `Y`, and `Z`) and their indexing mechanism (e.g., x_{ijk} and $[\mathsf{X}]_{1,2i-1,3}$) is similar to that of matrices.

Tensors will become more important when we start working with images, which arrive as n-dimensional arrays with 3 axes corresponding to the height, width, and a *channel* axis for stacking the color channels (red, green, and blue). For now, we will skip over higher order tensors and focus on the basics.

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

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

       [[12., 13., 14., 15.],
```

(continues on next page)

(continued from previous page)

```
        [16., 17., 18., 19.],
        [20., 21., 22., 23.]]])
```

2.3.5 Basic Properties of Tensor Arithmetic

Scalars, vectors, matrices, and tensors ("tensors" in this subsection refer to algebraic objects) of an arbitrary number of axes have some nice properties that often come in handy. For example, you might have noticed from the definition of an elementwise operation that any elementwise unary operation does not change the shape of its operand. Similarly, given any two tensors with the same shape, the result of any binary elementwise operation will be a tensor of that same shape. For example, adding two matrices of the same shape performs elementwise addition over these two matrices.

```
A = np.arange(20).reshape(5, 4)
B = A.copy()  # Assign a copy of `A` to `B` by allocating new memory
A, A + B
```

```
(array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]]),
 array([[ 0.,  2.,  4.,  6.],
        [ 8., 10., 12., 14.],
        [16., 18., 20., 22.],
        [24., 26., 28., 30.],
        [32., 34., 36., 38.]]))
```

Specifically, elementwise multiplication of two matrices is called their *Hadamard product* (math notation $\odot$). Consider matrix $\mathbf{B} \in \mathbb{R}^{m \times n}$ whose element of row i and column j is b_{ij}. The Hadamard product of matrices $\mathbf{A}$ (defined in (2.3.2)) and $\mathbf{B}$

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

```
A * B
```

```
array([[  0.,   1.,   4.,   9.],
       [ 16.,  25.,  36.,  49.],
       [ 64.,  81., 100., 121.],
       [144., 169., 196., 225.],
       [256., 289., 324., 361.]])
```

Multiplying or adding a tensor by a scalar also does not change the shape of the tensor, where each element of the operand tensor will be added or multiplied by the scalar.

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

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

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

2.3.6 Reduction

One useful operation that we can perform with arbitrary tensors is to calculate the sum of their elements. In mathematical notation, we express sums using the $\sum$ symbol. To express the sum of the elements in a vector $\mathbf{x}$ of length d, we write $\sum_{i=1}^{d} x_i$. In code, we can just call the function for calculating the sum.

```
x = np.arange(4)
x, x.sum()
```

```
(array([0., 1., 2., 3.]), array(6.))
```

We can express sums over the elements of tensors of arbitrary shape. 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()
```

```
((5, 4), array(190.))
```

By default, invoking the function for calculating the sum *reduces* a tensor along all its axes to a scalar. We can also specify the axes along which the tensor is reduced via summation. Take matrices as an example. To reduce the row dimension (axis 0) by summing up elements of all the rows, we specify `axis=0` when invoking the function. Since the input matrix reduces along axis 0 to generate the output vector, the dimension of axis 0 of the input is lost in the output shape.

```
A_sum_axis0 = A.sum(axis=0)
A_sum_axis0, A_sum_axis0.shape
```

```
(array([40., 45., 50., 55.]), (4,))
```

Specifying `axis=1` will reduce the column dimension (axis 1) by summing up elements of all the columns. Thus, the dimension of axis 1 of the input is lost in the output shape.

```
A_sum_axis1 = A.sum(axis=1)
A_sum_axis1, A_sum_axis1.shape
```

```
(array([ 6., 22., 38., 54., 70.]), (5,))
```

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])  # Same as `A.sum()`
```

```
array(190.)
```

A related quantity is the *mean*, which is also called the *average*. We calculate the mean by dividing the sum by the total number of elements. In code, we could just call the function for calculating the mean on tensors of arbitrary shape.

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

```
(array(9.5), array(9.5))
```

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

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

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

Non-Reduction Sum

However, sometimes it can be useful to keep the number of axes unchanged when invoking the function for calculating the sum or mean.

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

```
array([[ 6.],
       [22.],
       [38.],
       [54.],
       [70.]])
```

For instance, since `sum_A` still keeps its two axes after summing each row, we can divide `A` by `sum_A` with broadcasting.

```
A / sum_A
```

```
array([[0.        , 0.16666667, 0.33333334, 0.5       ],
       [0.18181819, 0.22727273, 0.27272728, 0.3181818 ],
       [0.21052632, 0.23684211, 0.2631579 , 0.28947368],
       [0.22222222, 0.24074075, 0.25925925, 0.2777778 ],
       [0.22857143, 0.24285714, 0.25714287, 0.27142859]])
```

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. This function will not reduce the input tensor along any axis.

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

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  6.,  8., 10.],
       [12., 15., 18., 21.],
       [24., 28., 32., 36.],
       [40., 45., 50., 55.]])
```

2.3.7 Dot Products

So far, we have only performed elementwise operations, sums, and averages. And if this was all we could do, linear algebra probably would not deserve its own section. However, one of the most fundamental operations is the dot product. Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their *dot product* $\mathbf{x}^\top \mathbf{y}$ (or $\langle \mathbf{x}, \mathbf{y} \rangle$) 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 = np.ones(4)
x, y, np.dot(x, y)
```

```
(array([0., 1., 2., 3.]), array([1., 1., 1., 1.]), array(6.))
```

Note that we can express the dot product of two vectors equivalently by performing an elementwise multiplication and then a sum:

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

```
array(6.)
```

Dot products are useful in a wide range of contexts. For example, given some set of values, denoted by a vector $\mathbf{x} \in \mathbb{R}^d$ and a set of weights denoted by $\mathbf{w} \in \mathbb{R}^d$, 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 non-negative and sum to one (i.e., $\left(\sum_{i=1}^{d} w_i = 1\right)$), the dot product expresses a *weighted average*. After normalizing two vectors to have the unit length, the dot products express the cosine of the angle between them. We will formally introduce this notion of *length* later in this section.

2.3.8 Matrix-Vector Products

Now that we know how to calculate dot products, we can begin to understand *matrix-vector products*. Recall the matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and the vector $\mathbf{x} \in \mathbb{R}^n$ defined and visualized in (2.3.2) and (2.3.1) respectively. Let us start off by visualizing the matrix $\mathbf{A}$ 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 $\mathbf{a}_i^\top \mathbf{x}$:

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

We can think of multiplication by 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 turn out to be remarkably useful. For example, we can represent rotations as multiplications by a square matrix. As we will see in subsequent chapters, we can also use matrix-vector products to describe the most intensive calculations required when computing each layer in a neural network given the values of the previous layer.

Expressing matrix-vector products in code with tensors, we use the same `dot` function as for dot products. When we call `np.dot(A, x)` with a matrix `A` and a vector `x`, the matrix-vector product is performed. Note that the column dimension of `A` (its length along axis 1) must be the same as the dimension of `x` (its length).

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

```
((5, 4), (4,), array([ 14.,  38.,  62.,  86., 110.]))
```

2.3.9 Matrix-Matrix Multiplication

If 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 $\mathbf{B} \in \mathbb{R}^{k \times m}$:

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

Denote by $\mathbf{a}_i^\top \in \mathbb{R}^k$ the row vector representing the i^{th} row of the matrix $\mathbf{A}$, and let $\mathbf{b}_j \in \mathbb{R}^k$ be the column vector from the j^{th} column of the matrix $\mathbf{B}$. To produce the matrix product $\mathbf{C} = \mathbf{AB}$, it is easiest to think of $\mathbf{A}$ in terms of its row vectors and $\mathbf{B}$ in terms of its column vectors:

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

Then the matrix product $\mathbf{C} \in \mathbb{R}^{n \times m}$ is produced as we simply compute each element c_{ij} as the dot product $\mathbf{a}_i^\top \mathbf{b}_j$:

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

We can think of the matrix-matrix multiplication **AB** as simply performing m matrix-vector 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 5 rows and 4 columns, and `B` is a matrix with 4 rows and 3 columns. After multiplication, we obtain a matrix with 5 rows and 3 columns.

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

```
array([[ 6.,  6.,  6.],
       [22., 22., 22.],
       [38., 38., 38.],
       [54., 54., 54.],
       [70., 70., 70.]])
```

Matrix-matrix multiplication can be simply called *matrix multiplication*, and should not be confused with the Hadamard product.

2.3.10 Norms

Some of the most useful operators in linear algebra are *norms*. Informally, the norm of a vector tells us how *big* a vector is. The notion of *size* under consideration here concerns not dimensionality but rather the magnitude of the components.

In linear algebra, a vector norm is a function f that maps a vector to a scalar, satisfying a handful of properties. Given any vector $\mathbf{x}$, the first property says that if we scale all the elements of a vector by a constant factor α, its norm also scales by the *absolute value* of the same constant factor:

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

The second property is the familiar triangle inequality:

$$f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}). \tag{2.3.11}$$

The third property simply says that the norm must be non-negative:

$$f(\mathbf{x}) \geq 0. \tag{2.3.12}$$

That makes sense, as in most contexts the smallest *size* for anything is 0. The final property requires that the smallest norm is achieved and only achieved by a vector consisting of all zeros.

$$\forall i, [\mathbf{x}]_i = 0 \Leftrightarrow f(\mathbf{x}) = 0. \tag{2.3.13}$$

You might notice that norms sound a lot like measures of distance. And if you remember Euclidean distances (think Pythagoras' theorem) from grade school, then the concepts of non-negativity and the triangle inequality might ring a bell. In fact, the Euclidean distance is a norm: specifically it is the L_2 norm. Suppose that the elements in the n-dimensional vector $\mathbf{x}$ are $x_1, \ldots, x_n$.

The L_2 *norm* of $\mathbf{x}$ is the square root of the sum of the squares of the vector elements:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}, \tag{2.3.14}$$

where the subscript 2 is often omitted in L_2 norms, i.e., $\|\mathbf{x}\|$ is equivalent to $\|\mathbf{x}\|_2$. In code, we can calculate the L_2 norm of a vector as follows.

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

```
array(5.)
```

In deep learning, we work more often with the squared L_2 norm.

You will also frequently encounter the L_1 *norm*, which is expressed as the sum of the absolute values of the vector elements:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| \,. \tag{2.3.15}$$

As compared with the L_2 norm, it is less influenced by outliers. To calculate the L_1 norm, we compose the absolute value function with a sum over the elements.

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

```
array(7.)
```

Both the L_2 norm and the L_1 norm are special cases of the more general L_p *norm*:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}. \tag{2.3.16}$$

Analogous to L_2 norms of vectors, the *Frobenius norm* of a matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ is the square root of the sum of the squares of the matrix elements:

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}^2}. \tag{2.3.17}$$

The Frobenius norm satisfies all the properties of vector norms. It 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.

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

```
array(6.)
```

Norms and Objectives

While we do not want to get too far ahead of ourselves, we can plant some intuition already about why these concepts are useful. In deep learning, we are often trying to solve optimization problems: *maximize* the probability assigned to observed data; *minimize* the distance between predictions and the ground-truth observations. Assign vector representations to items (like words, products, or news articles) such that the distance between similar items is minimized, and the distance between dissimilar items is maximized. Oftentimes, the objectives, perhaps the most important components of deep learning algorithms (besides the data), are expressed as norms.

2.3.11 More on Linear Algebra

In just this section, we have taught you all the linear algebra that you will need to understand a remarkable chunk of modern deep learning. There is a lot more to linear algebra and a lot of that mathematics 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 much more inclined to learn more mathematics once you have gotten your hands dirty deploying useful machine learning models on real datasets. So while we reserve the right to introduce more mathematics much later on, we will wrap up this section here.

If you are eager to learn more about linear algebra, you may refer to either the online appendix on linear algebraic operations[41] or other excellent resources (Strang, 1993; Kolter, 2008; Petersen et al., 2008).

Summary

- Scalars, vectors, matrices, and tensors are basic mathematical objects in linear algebra.
- Vectors generalize scalars, and matrices generalize vectors.
- Scalars, vectors, matrices, and tensors have zero, one, two, and an arbitrary number of axes, respectively.
- A tensor can be reduced along the specified axes by `sum` and `mean`.
- Elementwise multiplication of two matrices is called their Hadamard product. It is different from matrix multiplication.
- In deep learning, we often work with norms such as the L_1 norm, the L_2 norm, and the Frobenius norm.
- We can perform a variety of operations over scalars, vectors, matrices, and tensors.

Exercises

1. Prove that the transpose of a matrix $\mathbf{A}$'s transpose is $\mathbf{A}$: $(\mathbf{A}^\top)^\top = \mathbf{A}$.
2. Given two matrices $\mathbf{A}$ and $\mathbf{B}$, show that the sum of transposes is equal to the transpose of a sum: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$.
3. Given any square matrix $\mathbf{A}$, is $\mathbf{A} + \mathbf{A}^\top$ always symmetric? Why?
4. We defined the tensor `X` of shape (2, 3, 4) in this section. What is the output of `len(X)`?
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 reason?
7. When traveling between two points in 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?

[41] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/geometry-linear-algebraic-ops.html

8. Consider a tensor with shape (2, 3, 4). What are the shapes of the summation outputs along axis 0, 1, and 2?
9. Feed a tensor with 3 or more axes to the `linalg.norm` function and observe its output. What does this function compute for tensors of arbitrary shape?

Discussions[42]

2.4 Calculus

Finding the area of a polygon had remained mysterious until at least 2,500 years ago, when ancient Greeks divided a polygon into triangles and summed their areas. To find the area of curved shapes, such as a circle, ancient Greeks inscribed polygons in such shapes. As shown in Fig. 2.4.1, an inscribed polygon with more sides of equal length better approximates the circle. This process is also known as the *method of exhaustion*.

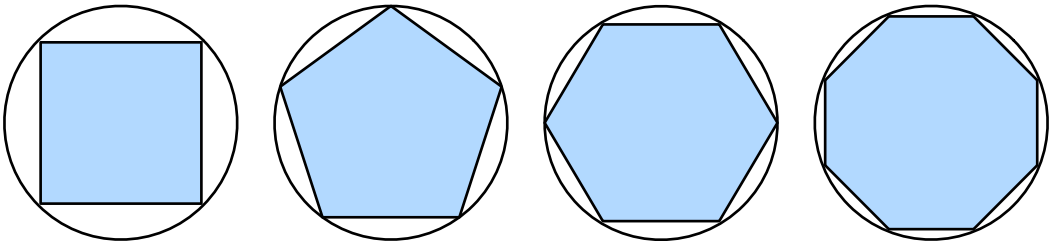


Fig. 2.4.1: Find the area of a circle with the method of exhaustion.

In fact, the method of exhaustion is where *integral calculus* (will be described in Section 18.5) originates from. More than 2,000 years later, the other branch of calculus, *differential calculus*, was invented. Among the most critical applications of differential calculus, optimization problems consider how to do something *the best*. As discussed in Section 2.3.10, such problems are ubiquitous in deep learning.

In deep learning, we *train* models, updating them successively so that they get better and better as they see more and more data. Usually, getting better means minimizing a *loss function*, a score that answers the question "how *bad* is our model?" This question is more subtle than it appears. Ultimately, what we really care about is producing a model that performs well on data that we have never seen before. But we can only fit the model to data that we can actually see. Thus we can decompose the task of fitting models into two key concerns: i) *optimization*: the process of fitting our models to observed data; ii) *generalization*: the mathematical principles and practitioners' wisdom that guide as to how to produce models whose validity extends beyond the exact set of data examples used to train them.

To help you understand optimization problems and methods in later chapters, here we give a very brief primer on differential calculus that is commonly used in deep learning.

[42] https://discuss.d2l.ai/t/30

2.4.1 Derivatives and Differentiation

We begin by addressing the calculation of derivatives, a crucial step in nearly all deep learning optimization algorithms. In deep learning, we typically choose loss functions that are differentiable with respect to our model's parameters. Put simply, this means that for each parameter, we can determine how rapidly the loss would increase or decrease, were we to *increase* or *decrease* that parameter by an infinitesimally small amount.

Suppose that we have a function $f : \mathbb{R} \to \mathbb{R}$, whose input and output are both scalars. The *derivative* of f is defined as

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

if this limit exists. If $f'(a)$ exists, f is said to be *differentiable* at a. If f is differentiable at every number of an interval, then this function is differentiable on this interval. We can interpret the derivative $f'(x)$ in (2.4.1) as the *instantaneous* rate of change of $f(x)$ with respect to x. The so-called instantaneous rate of change is based on the variation h in x, which approaches 0.

To illustrate derivatives, let us experiment with an example. Define $u = f(x) = 3x^2 - 4x$.

```
%matplotlib inline
from IPython import display
from mxnet import np, npx
from d2l import mxnet as d2l

npx.set_np()

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

By setting $x = 1$ and letting h approach 0, the numerical result of $\frac{f(x+h) - f(x)}{h}$ in (2.4.1) approaches 2. Though this experiment is not a mathematical proof, we will see later that the derivative u' is 2 when $x = 1$.

```
def numerical_lim(f, x, h):
    return (f(x + h) - f(x)) / h

h = 0.1
for i in range(5):
    print(f'h={h:.5f}, numerical limit={numerical_lim(f, 1, h):.5f}')
    h *= 0.1
```

```
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
```

Let us familiarize ourselves with a few equivalent notations for derivatives. Given $y = f(x)$, where x and y are the independent variable and the dependent variable of the function f, respectively. 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 symbols $\frac{d}{dx}$ and D are *differentiation operators* that indicate operation of *differentiation*. We can use the following rules to differentiate common functions:

- $DC = 0$ (C is a constant),
- $Dx^n = nx^{n-1}$ (the *power rule*, n is any real number),
- $De^x = e^x$,
- $D\ln(x) = 1/x$.

To differentiate a function that is formed from a few simpler functions such as the above common functions, the following rules can be handy for us. Suppose that functions f and g are both differentiable and C is a constant, we have the *constant multiple rule*

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x), \tag{2.4.3}$$

the *sum rule*

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \tag{2.4.4}$$

the *product rule*

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \tag{2.4.5}$$

and the *quotient rule*

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \tag{2.4.6}$$

Now we can apply a few of the above rules to find $u' = f'(x) = 3\frac{d}{dx}x^2 - 4\frac{d}{dx}x = 6x - 4$. Thus, by setting $x = 1$, we have $u' = 2$: this is supported by our earlier experiment in this section where the numerical result approaches 2. This derivative is also the slope of the tangent line to the curve $u = f(x)$ when $x = 1$.

To visualize such an interpretation of derivatives, we will use `matplotlib`, a popular plotting library in Python. To configure properties of the figures produced by `matplotlib`, we need to define a few functions. In the following, the `use_svg_display` function specifies the `matplotlib` package to output the svg figures for sharper images. Note that the comment `#@save` is a special mark where the following function, class, or statements are saved in the `d2l` package so later they can be directly invoked (e.g., `d2l.use_svg_display()`) without being redefined.

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

We define the `set_figsize` function to specify the figure sizes. Note that here we directly use `d2l.plt` since the import statement `from matplotlib import pyplot as plt` has been marked for being saved in the `d2l` package in the preface.

```
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 following `set_axes` function sets properties of axes of figures produced by `matplotlib`.

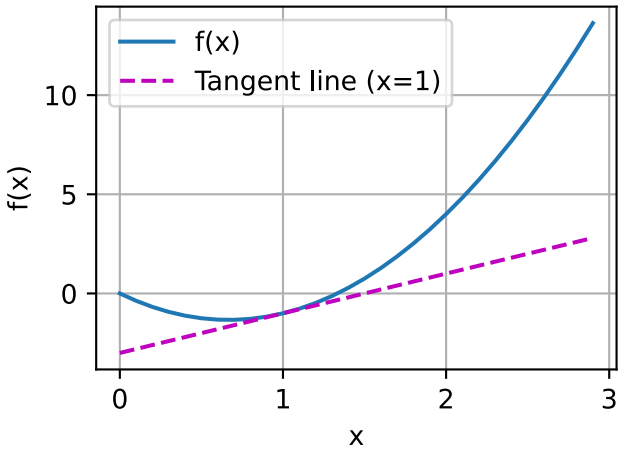
```
#@save
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """Set the axes for matplotlib."""
    axes.set_xlabel(xlabel)
    axes.set_ylabel(ylabel)
    axes.set_xscale(xscale)
    axes.set_yscale(yscale)
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    if legend:
   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

With these three functions for figure configurations, we define the `plot` function to plot multiple curves succinctly since we will need to visualize many curves throughout the book.

```
#@save
def plot(X, Y=None, xlabel=None, ylabel=None, legend=None, xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None):
    """Plot data points."""
    if legend is None:
        legend = []

    set_figsize(figsize)
    axes = axes if axes else d2l.plt.gca()

    # Return True if `X` (tensor or list) has 1 axis
    def has_one_axis(X):
        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)
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        if len(x):
            axes.plot(x, y, fmt)
        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)'])
```



2.4.2 Partial Derivatives

So far we have dealt with the differentiation of functions of just one variable. In deep learning, functions often depend on *many* variables. Thus, we need to extend the ideas of differentiation to these *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

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{2.4.7}$$

To calculate $\frac{\partial y}{\partial x_i}$, we can simply 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. For notation of partial derivatives, the following are equivalent:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f. \tag{2.4.8}$$

2.4.3 Gradients

We can concatenate partial derivatives of a multivariate function with respect to all its variables to obtain the *gradient* vector 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 $f(\mathbf{x})$ with respect to $\mathbf{x}$ is a vector of n partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right]^\top, \tag{2.4.9}$$

where $\nabla_{\mathbf{x}} f(\mathbf{x})$ is often replaced by $\nabla f(\mathbf{x})$ when there is no ambiguity.

Let $\mathbf{x}$ be an n-dimensional vector, the following rules are often used when differentiating multivariate functions:

- For all $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$,
- For all $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$,
- For all $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$,
- $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$.

Similarly, for any matrix $\mathbf{X}$, we have $\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$. As we will see later, gradients are useful for designing optimization algorithms in deep learning.

2.4.4 Chain Rule

However, such gradients can be hard to find. This is because multivariate functions in deep learning are often *composite*, so we may not apply any of the aforementioned rules to differentiate these functions. Fortunately, the *chain rule* enables us to differentiate composite functions.

Let us first consider functions of a single variable. Suppose that functions $y = f(u)$ and $u = g(x)$ are both differentiable, then the chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}. \tag{2.4.10}$$

Now let us turn our attention to a more general scenario where functions have an arbitrary number of variables. Suppose that the differentiable function y has variables $u_1, u_2, \ldots, u_m$, where each differentiable function u_i has variables $x_1, x_2, \ldots, x_n$. Note that y is a function of $x_1, x_2, \ldots, x_n$. Then the chain rule gives

$$\frac{dy}{dx_i} = \frac{dy}{du_1}\frac{du_1}{dx_i} + \frac{dy}{du_2}\frac{du_2}{dx_i} + \cdots + \frac{dy}{du_m}\frac{du_m}{dx_i} \tag{2.4.11}$$

for any $i = 1, 2, \ldots, n$.

Summary

- Differential calculus and integral calculus are two branches of calculus, where the former can be applied to the ubiquitous optimization problems in deep learning.
- A derivative can be interpreted as the instantaneous rate of change of a function with respect to its variable. It is also the slope of the tangent line to the curve of the function.
- A gradient is a vector whose components are the partial derivatives of a multivariate function with respect to all its variables.
- The chain rule enables us to differentiate composite functions.

Exercises

1. Plot the function $y = f(x) = x^3 - \frac{1}{x}$ and its tangent line when $x = 1$.
2. Find the gradient of the function $f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$.
3. What is the gradient of the function $f(\mathbf{x}) = \|\mathbf{x}\|_2$?
4. 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 $z = z(a, b)$?

Discussions[43]

[43] https://discuss.d2l.ai/t/32

2.5 Automatic Differentiation

As we have explained in Section 2.4, differentiation is a crucial step in nearly all deep learning optimization algorithms. While the calculations for taking these derivatives are straightforward, requiring only some basic calculus, for complex models, working out the updates by hand can be a pain (and often error-prone).

Deep learning frameworks expedite this work by automatically calculating derivatives, i.e., *automatic differentiation*. In practice, based on our designed model the system builds a *computational graph*, tracking which data combined through which operations to produce the output. Automatic differentiation enables the system to subsequently backpropagate gradients. Here, *backpropagate* simply means to trace through the computational graph, filling in the partial derivatives with respect to each parameter.

2.5.1 A Simple Example

As a toy example, say 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, let us create the variable `x` and assign it an initial value.

```
from mxnet import autograd, np, npx

npx.set_np()

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

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

Before we even calculate the gradient of y with respect to $\mathbf{x}$, we will need a place to store it. It is important that we do not allocate new memory every time we take a derivative with respect to a parameter because we will often update the same parameters thousands or millions of times and could quickly run out of memory. Note that a gradient of a scalar-valued function with respect to a vector $\mathbf{x}$ is itself vector-valued and has the same shape as $\mathbf{x}$.

```
# We allocate memory for a tensor's gradient by invoking `attach_grad`
x.attach_grad()
# After we calculate a gradient taken with respect to `x`, we will be able to
# access it via the `grad` attribute, whose values are initialized with 0s
x.grad
```

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

Now let us calculate y.

```
# Place our code inside an `autograd.record` scope to build the computational
# graph
with autograd.record():
    y = 2 * np.dot(x, x)
y
```

```
array(28.)
```

Since x is a vector of length 4, an inner product of x and x is performed, yielding the scalar output that we assign to y. Next, we can automatically calculate the gradient of y with respect to each component of x by calling the function for backpropagation and printing the gradient.

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

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

The gradient of the function $y = 2\mathbf{x}^\top\mathbf{x}$ with respect to $\mathbf{x}$ should be $4\mathbf{x}$. Let us quickly verify that our desired gradient was calculated correctly.

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

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

Now let us calculate another function of x.

```
with autograd.record():
    y = x.sum()
y.backward()
x.grad  # Overwritten by the newly calculated gradient
```

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

2.5.2 Backward for Non-Scalar Variables

Technically, when y is not a scalar, the most natural interpretation of the differentiation of a vector y with respect to a vector x is a matrix. For higher-order and higher-dimensional y and x, the differentiation result could be a high-order tensor.

However, while these more exotic objects do show up in advanced machine learning (including in deep learning), more often when we are calling backward on a vector, we are trying to calculate the derivatives of the loss functions for each constituent of a *batch* of training examples. Here, our intent is not to calculate the differentiation matrix but rather the sum of the partial derivatives computed individually for each example in the batch.

```
# When we invoke `backward` on a vector-valued variable `y` (function of `x`),
# a new scalar variable is created by summing the elements in `y`. Then the
# gradient of that scalar variable with respect to `x` is computed
with autograd.record():
    y = x * x  # `y` is a vector
y.backward()
x.grad  # Equals to y = sum(x * x)
```

```
array([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 `y` was calculated as a function of `x`, and that subsequently `z` was calculated as a function of both `y` and `x`. Now, imagine that we wanted to calculate the gradient of `z` with respect to `x`, but wanted for some reason to treat `y` as a constant, and only take into account the role that `x` played after `y` was calculated.

Here, we can detach `y` to return a new variable `u` that has the same value as `y` but discards any information about how `y` was computed in the computational graph. In other words, the gradient will not flow backwards through `u` to `x`. Thus, the following backpropagation function computes the partial derivative of `z = u * x` with respect to `x` while treating `u` as a constant, instead of the partial derivative of `z = x * x * x` with respect to `x`.

```
with autograd.record():
    y = x * x
    u = y.detach()
    z = u * x
z.backward()
x.grad == u
```

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

Since the computation of `y` was recorded, we can subsequently invoke backpropagation on `y` to get the derivative of `y = x * x` with respect to `x`, which is `2 * x`.

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

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

2.5.4 Computing the Gradient of Python Control Flow

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. In the following snippet, note that 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 np.linalg.norm(b) < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

Let us compute the gradient.

```
a = np.random.normal()
a.attach_grad()
with autograd.record():
    d = f(a)
d.backward()
```

We can now analyze the `f` function defined above. Note that it is piecewise linear in its input `a`. In other words, for any `a` there exists some constant scalar `k` such that `f(a) = k * a`, where the value of `k` depends on the input `a`. Consequently `d / a` allows us to verify that the gradient is correct.

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

```
array(True)
```

Summary

- Deep learning frameworks can automate the calculation of derivatives. To use it, we first attach gradients to those variables with respect to which we desire partial derivatives. We then record the computation of our target value, execute its function for backpropagation, and access the resulting gradient.

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.
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 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. Redesign an example of finding the gradient of the control flow. Run and analyze the result.
5. Let $f(x) = \sin(x)$. Plot $f(x)$ and $\frac{df(x)}{dx}$, where the latter is computed without exploiting that $f'(x) = \cos(x)$.

Discussions[44]

[44] https://discuss.d2l.ai/t/34

2.6 Probability

In some form or another, machine learning is all about making predictions. We might want to predict the *probability* of a patient suffering a heart attack in the next year, given their clinical history. In anomaly detection, we might want to assess how *likely* a set of readings from an airplane's jet engine would be, were it operating normally. In reinforcement learning, we want an agent to act intelligently in an environment. This means we need to think about the probability of getting a high reward under each of the available actions. And when we build recommender systems we also need to think about probability. For example, say *hypothetically* that we worked for a large online bookseller. We might want to estimate the probability that a particular user would buy a particular book. For this we need to use the language of probability. Entire courses, majors, theses, careers, and even departments, are devoted to probability. So naturally, our goal in this section is not to teach the whole subject. Instead we hope to get you off the ground, to teach you just enough that you can start building your first deep learning models, and to give you enough of a flavor for the subject that you can begin to explore it on your own if you wish.

We have already invoked probabilities in previous sections without articulating what precisely they are or giving a concrete example. Let us get more serious now by considering the first case: distinguishing cats and dogs based on photographs. This might sound simple but it is actually a formidable challenge. To start with, the difficulty of the problem may depend on the resolution of the image.

Fig. 2.6.1: Images of varying resolutions (10×10, 20×20, 40×40, 80×80, and 160×160 pixels).

As shown in Fig. 2.6.1, while it is easy for humans to recognize cats and dogs at the resolution of 160×160 pixels, it becomes challenging at 40×40 pixels and next to impossible at 10×10 pixels. In other words, our ability to tell cats and dogs apart at a large distance (and thus low resolution) might approach uninformed guessing. Probability gives us a formal way of reasoning about our level of certainty. If we are completely sure that the image depicts a cat, we say that the *probability* that the corresponding label y is "cat", denoted $P(y = \text{“cat”})$ equals 1. If we had no evidence to suggest that $y =$ "cat" or that $y =$ "dog", then we might say that the two possibilities were equally *likely* expressing this as $P(y = \text{“cat”}) = P(y = \text{“dog”}) = 0.5$. If we were reasonably confident, but not sure that the image depicted a cat, we might assign a probability $0.5 < P(y = \text{“cat”}) < 1$.

Now consider the second case: given some weather monitoring data, we want to predict the probability that it will rain in Taipei tomorrow. If it is summertime, the rain might come with probability 0.5.

In both cases, we have some value of interest. And in both cases we are uncertain about the outcome. But there is a key difference between the two cases. In this first case, the image is in fact either a dog or a cat, and we just do not know which. In the second case, the outcome may actually be a random event, if you believe in such things (and most physicists do). So probability is a flexible language for reasoning about our level of certainty, and it can be applied effectively in a broad set of contexts.

2.6.1 Basic Probability Theory

Say that we cast a die and want to know what the chance is of seeing a 1 rather than another digit. If the die is fair, all the six outcomes $\{1, \ldots, 6\}$ are equally likely to occur, and thus we would see a 1 in one out of six cases. Formally we state that 1 occurs with probability $\frac{1}{6}$.

For a real die that we receive from a factory, we might not know those proportions and we would need to check whether it is tainted. The only way to investigate the die is by casting it many times and recording the outcomes. For each cast of the die, we will observe a value in $\{1, \ldots, 6\}$. Given these outcomes, we want to investigate the probability of observing each outcome.

One natural approach for each value is to take the individual count for that value and to divide it by the total number of tosses. This gives us an *estimate* of the probability of a given *event*. The *law of large numbers* tell us that as the number of tosses grows this estimate will draw closer and closer to the true underlying probability. Before going into the details of what is going here, let us try it out.

To start, let us import the necessary packages.

```
%matplotlib inline
import random
from mxnet import np, npx
from d2l import mxnet as d2l

npx.set_np()
```

Next, we will want to be able to cast the die. In statistics we call this process of drawing examples from probability distributions *sampling*. The distribution that assigns probabilities to a number of discrete choices is called the *multinomial distribution*. We will give a more formal definition of *distribution* later, but at a high level, think of it as just an assignment of probabilities to events.

To draw a single sample, we simply pass in a vector of probabilities. The output is another vector of the same length: its value at index i is the number of times the sampling outcome corresponds to i.

```
fair_probs = [1.0 / 6] * 6
np.random.multinomial(1, fair_probs)
```

```
array([0, 0, 0, 1, 0, 0], dtype=int64)
```

If you run the sampler a bunch of times, you will find that you get out random values each time. As with estimating the fairness of a die, we often want to generate many samples from the same

distribution. It would be unbearably slow to do this with a Python `for` loop, so the function we are using supports drawing multiple samples at once, returning an array of independent samples in any shape we might desire.

```
np.random.multinomial(10, fair_probs)
```

```
array([1, 1, 5, 1, 1, 1], dtype=int64)
```

Now that we know how to sample rolls of a die, we can simulate 1000 rolls. We can then go through and count, after each of the 1000 rolls, how many times each number was rolled. Specifically, we calculate the relative frequency as the estimate of the true probability.

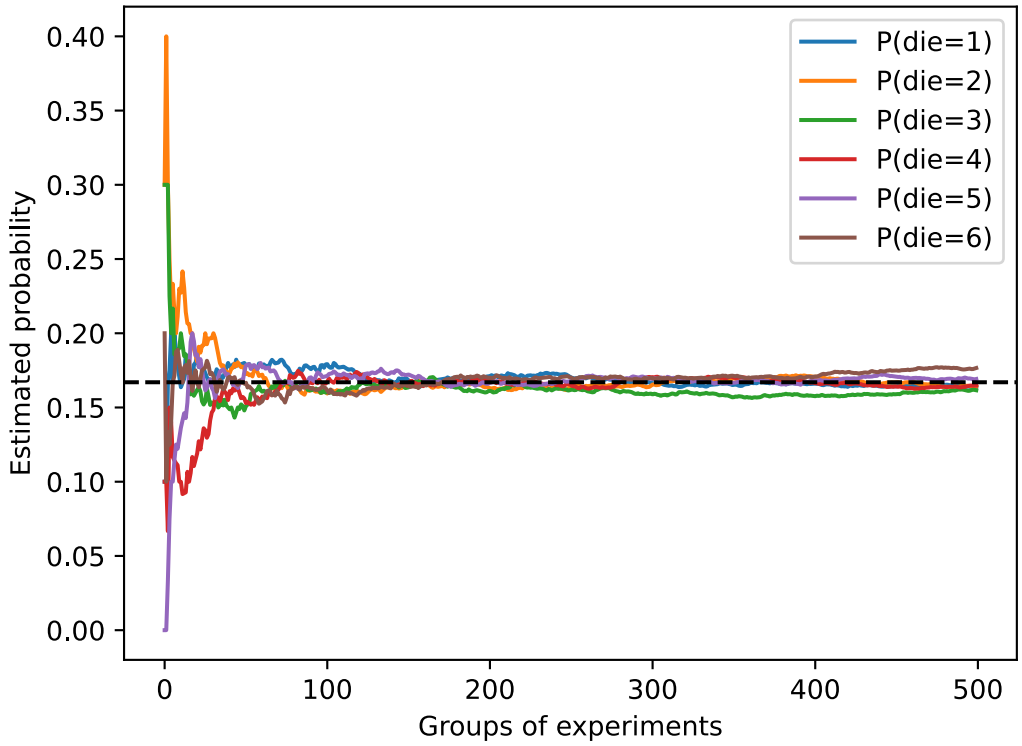
```
counts = np.random.multinomial(1000, fair_probs).astype(np.float32)
counts / 1000
```

```
array([0.162, 0.149, 0.178, 0.17 , 0.166, 0.175])
```

Because we generated the data from a fair die, we know that each outcome has true probability $\frac{1}{6}$, roughly 0.167, so the above output estimates look good.

We can also visualize how these probabilities converge over time towards the true probability. Let us conduct 500 groups of experiments where each group draws 10 samples.

```
counts = np.random.multinomial(10, fair_probs, size=500)
cum_counts = counts.astype(np.float32).cumsum(axis=0)
estimates = cum_counts / cum_counts.sum(axis=1, keepdims=True)

d2l.set_figsize((6, 4.5))
for i in range(6):
    d2l.plt.plot(estimates[:, i].asnumpy(),
                 label=("P(die=" + str(i + 1) + ")"))
d2l.plt.axhline(y=0.167, color='black', linestyle='dashed')
d2l.plt.gca().set_xlabel('Groups of experiments')
d2l.plt.gca().set_ylabel('Estimated probability')
d2l.plt.legend();
```



Each solid curve corresponds to one of the six values of the die and gives our estimated probability that the die turns up that value as assessed after each group of experiments. The dashed black line gives the true underlying probability. As we get more data by conducting more experiments, the 6 solid curves converge towards the true probability.

Axioms of Probability Theory

When dealing with the rolls of a die, we call the set $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ the *sample space* or *outcome space*, where each element is an *outcome*. An *event* is a set of outcomes from a given sample space. For instance, "seeing a 5" ($\{5\}$) and "seeing an odd number" ($\{1, 3, 5\}$) are both valid events of rolling a die. Note that if the outcome of a random experiment is in event $\mathcal{A}$, then event $\mathcal{A}$ has occurred. That is to say, if 3 dots faced up after rolling a die, since $3 \in \{1, 3, 5\}$, we can say that the event "seeing an odd number" has occurred.

Formally, *probability* can be thought of as a function that maps a set to a real value. The probability of an event $\mathcal{A}$ in the given sample space $\mathcal{S}$, denoted as $P(\mathcal{A})$, satisfies the following properties:

- For any event $\mathcal{A}$, its probability is never negative, i.e., $P(\mathcal{A}) \geq 0$;
- 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* ($\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ for all $i \neq j$), the probability that any 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 are also the axioms of probability theory, proposed by Kolmogorov in 1933. Thanks to this axiom system, we can avoid any philosophical dispute on randomness; instead, we can reason rigorously with a mathematical language. For instance, by letting event $\mathcal{A}_1$ be the entire sample space and $\mathcal{A}_i = \emptyset$ for all $i > 1$, we can prove that $P(\emptyset) = 0$, i.e., the probability of an impossible event is 0.

In our random experiment of casting a die, we introduced the notion of a *random variable*. A random variable can be pretty much any quantity and is not deterministic. It could take one value among a set of possibilities in a random experiment. Consider a random variable X whose value is in the sample space $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ of rolling a die. We can denote the event "seeing a 5" as $\{X = 5\}$ or $X = 5$, and its probability as $P(\{X = 5\})$ or $P(X = 5)$. By $P(X = a)$, we make a distinction between the random variable X and the values (e.g., a) that X can take. However, such pedantry results in a cumbersome notation. For a compact notation, on one hand, we can just denote $P(X)$ as the *distribution* over the random variable X: the distribution tells us the probability that X takes any value. On the other hand, we can simply write $P(a)$ to denote the probability that a random variable takes the value a. 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 $\{1 \leq X \leq 3\}$, which means $\{X = 1, 2, \text{or}, 3\}$. Equivalently, $P(1 \leq X \leq 3)$ represents the probability that the random variable X can take a value from $\{1, 2, 3\}$.

Note that there is a subtle difference between *discrete* random variables, like the sides of a die, and *continuous* ones, like the weight and the height of a person. There is little point in asking whether two people have exactly the same height. If we take precise enough measurements you will find that no two people on the planet have the exact same height. In fact, if we take a fine enough measurement, you will not have the same height when you wake up and when you go to sleep. So there is no purpose in asking about the probability that someone is 1.80139278291028719210196740527486202 meters tall. Given the world population of humans the probability is virtually 0. It makes more sense in this case to ask whether someone's height falls into a given interval, say between 1.79 and 1.81 meters. In these cases we quantify the likelihood that we see a value as a *density*. The height of exactly 1.80 meters has no probability, but nonzero density. In the interval between any two different heights we have nonzero probability. In the rest of this section, we consider probability in discrete space. For probability over continuous random variables, you may refer to Section 18.6.

2.6.2 Dealing with Multiple Random Variables

Very often, we will want to consider more than one random variable at a time. For instance, we may want to model the relationship between diseases and symptoms. Given a disease and a symptom, say "flu" and "cough", either may or may not occur in a patient with some probability. While we hope that the probability of both would be close to zero, we may want to estimate these probabilities and their relationships to each other so that we may apply our inferences to effect better medical care.

As a more complicated example, images contain millions of pixels, thus millions of random variables. And in many cases images will come with a label, identifying objects in the image. We can also think of the label as a random variable. We can even think of all the metadata as random variables such as location, time, aperture, focal length, ISO, focus distance, and camera type. All of these are random variables that occur jointly. When we deal with multiple random variables, there are several quantities of interest.

Joint Probability

The first is called the *joint probability* $P(A = a, B = b)$. Given any values a and b, the joint probability lets us answer, what is the probability that $A = a$ and $B = b$ simultaneously? Note that for any values a and b, $P(A = a, B = b) \leq P(A = a)$. This has to be the case, since for $A = a$ and $B = b$ to happen, $A = a$ has to happen *and* $B = b$ also has to happen (and vice versa). Thus, $A = a$ and $B = b$ cannot be more likely than $A = a$ or $B = b$ individually.

Conditional Probability

This brings us to an interesting ratio: $0 \leq \frac{P(A=a,B=b)}{P(A=a)} \leq 1$. We call this ratio a *conditional probability* and denote it by $P(B = b \mid A = a)$: it is the probability of $B = b$, provided that $A = a$ has occurred.

Bayes' theorem

Using the definition of conditional probabilities, we can derive one of the most useful and celebrated equations in statistics: *Bayes' theorem*. It goes as follows. By construction, we have the *multiplication rule* that $P(A, B) = P(B \mid A)P(A)$. By symmetry, this also holds for $P(A, B) = P(A \mid B)P(B)$. Assume that $P(B) > 0$. Solving for one of the conditional variables we get

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}. \tag{2.6.1}$$

Note that here we use the more compact notation where $P(A, B)$ is a *joint distribution* and $P(A \mid B)$ is a *conditional distribution*. Such distributions can be evaluated for particular values $A = a, B = b$.

Marginalization

Bayes' theorem is very useful if we want to infer one thing from the other, say cause and effect, but we only know the properties in the reverse direction, as we will see later in this section. One important operation that we need, to make this work, is *marginalization*. It is the operation of determining $P(B)$ from $P(A, B)$. We can see that the probability of B amounts to accounting for all possible choices of A and aggregating the joint probabilities over all of them:

$$P(B) = \sum_{A} P(A, B), \tag{2.6.2}$$

which is also known as the *sum rule*. The probability or distribution as a result of marginalization is called a *marginal probability* or a *marginal distribution*.

Independence

Another useful property to check for is *dependence* vs. *independence*. Two random variables A and B being independent means that the occurrence of one event of A does not reveal any information about the occurrence of an event of B. In this case $P(B \mid A) = P(B)$. Statisticians typically express this as $A \perp B$. From Bayes' theorem, it follows immediately that also $P(A \mid B) = P(A)$. In all the other cases we call A and B dependent. For instance, two successive rolls of a die are independent. In contrast, the position of a light switch and the brightness in the room are not

(they are not perfectly deterministic, though, since we could always have a broken light bulb, power failure, or a broken switch).

Since $P(A \mid B) = \frac{P(A,B)}{P(B)} = P(A)$ is equivalent to $P(A, B) = P(A)P(B)$, two random variables are independent if and only if their joint distribution is the product of their individual distributions. Likewise, two random variables A and B are *conditionally independent* given another random variable C if and only if $P(A, B \mid C) = P(A \mid C)P(B \mid C)$. This is expressed as $A \perp B \mid C$.

Application

Let us put our skills to the test. Assume that a doctor administers an HIV test to a patient. This test is fairly accurate and it fails only with 1% probability if the patient is healthy but reporting him as diseased. Moreover, it never fails to detect HIV if the patient actually has it. We use D_1 to indicate the diagnosis (1 if positive and 0 if negative) and H to denote the HIV status (1 if positive and 0 if negative). Table 2.6.1 lists such conditional probabilities.

Table 2.6.1: Conditional probability of $P(D_1 \mid H)$.

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 are not), since the conditional probability needs to sum up to 1, just like the probability. Let us work out the probability of the patient having HIV if the test comes back positive, i.e., $P(H = 1 \mid D_1 = 1)$. Obviously this is going to depend on how common the disease is, since it affects the number of false alarms. Assume that the population is quite healthy, e.g., $P(H = 1) = 0.0015$. To apply Bayes' theorem, we need to apply marginalization and the multiplication rule to determine

$$\begin{aligned} & P(D_1 = 1) \\ =& P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1) \\ =& P(D_1 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1 \mid H = 1)P(H = 1) \\ =& 0.011485. \end{aligned} \tag{2.6.3}$$

Thus, we get

$$\begin{aligned} & P(H = 1 \mid D_1 = 1) \\ =& \frac{P(D_1 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1)}. \\ =& 0.1306 \end{aligned} \tag{2.6.4}$$

In other words, there is only a 13.06% chance that the patient actually has HIV, despite using a very accurate test. 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, as shown in Table 2.6.2.

Table 2.6.2: Conditional probability of $P(D_2 \mid H)$.

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 us work out the requisite probabilities to invoke Bayes' theorem by assuming the conditional independence:

$$\begin{aligned}&P(D_1 = 1, D_2 = 1 \mid H = 0)\\ =&P(D_1 = 1 \mid H = 0)P(D_2 = 1 \mid H = 0)\\ =&0.0003,\end{aligned} \tag{2.6.5}$$

$$\begin{aligned}&P(D_1 = 1, D_2 = 1 \mid H = 1)\\ =&P(D_1 = 1 \mid H = 1)P(D_2 = 1 \mid H = 1)\\ =&0.98.\end{aligned} \tag{2.6.6}$$

Now we can apply marginalization and the multiplication rule:

$$\begin{aligned}&P(D_1 = 1, D_2 = 1)\\ =&P(D_1 = 1, D_2 = 1, H = 0) + P(D_1 = 1, D_2 = 1, H = 1)\\ =&P(D_1 = 1, D_2 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1)\\ =&0.00176955.\end{aligned} \tag{2.6.7}$$

In the end, the probability of the patient having HIV given both positive tests is

$$\begin{aligned}&P(H = 1 \mid D_1 = 1, D_2 = 1)\\ =&\frac{P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1, D_2 = 1)}\\ =&0.8307.\end{aligned} \tag{2.6.8}$$

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.

2.6.3 Expectation and Variance

To summarize key characteristics of probability distributions, we need some measures. The *expectation* (or average) of the random variable X is denoted as

$$E[X] = \sum_x xP(X = x). \tag{2.6.9}$$

When the input of a function $f(x)$ is a random variable drawn from the distribution P with different values x, the expectation of $f(x)$ is computed as

$$E_{x \sim P}[f(x)] = \sum_x f(x)P(x). \tag{2.6.10}$$

In many cases we want to measure by how much the random variable X deviates from its expectation. This can be quantified by the variance

$$\text{Var}[X] = E\left[(X - E[X])^2\right] = E[X^2] - E[X]^2. \tag{2.6.11}$$

Its square root is called the *standard deviation*. The variance of a function of a random variable measures by how much the function deviates from the expectation of the function, as different values x of the random variable are sampled from its distribution:

$$\text{Var}[f(x)] = E\left[\left(f(x) - E[f(x)]\right)^2\right]. \tag{2.6.12}$$

Summary

- We can sample from probability distributions.
- We can analyze multiple random variables using joint distribution, conditional distribution, Bayes' theorem, marginalization, and independence assumptions.
- Expectation and variance offer useful measures to summarize key characteristics of probability distributions.

Exercises

1. We conducted $m = 500$ groups of experiments where each group draws $n = 10$ samples. Vary m and n. Observe and analyze the experimental results.
2. 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: display the situation using a Venn Diagram[45].)
3. 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[46].)
4. In Section 2.6.2, the first test is more accurate. Why not run the first test twice rather than run both the first and second tests?

Discussions[47]

2.7 Documentation

Due to constraints on the length of this book, we cannot possibly introduce every single MXNet function and class (and you probably would not want us to). The API documentation and additional tutorials and examples provide plenty of documentation beyond the book. In this section we provide you with some guidance to exploring the MXNet API.

2.7.1 Finding All the Functions and Classes in a Module

In order 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:

```
from mxnet import np

print(dir(np.random))
```

```
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
↪ '__package__', '__spec__', '_mx_nd_np', 'beta', 'chisquare', 'choice', 'exponential',
↪'gamma', 'gumbel', 'logistic', 'lognormal', 'multinomial', 'multivariate_normal', 'normal',
↪ 'pareto', 'power', 'rand', 'randint', 'randn', 'rayleigh', 'shuffle', 'uniform', 'weibull
↪']
```

[45] https://en.wikipedia.org/wiki/Venn_diagram
[46] https://en.wikipedia.org/wiki/Markov_chain
[47] https://discuss.d2l.ai/t/36

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 Finding the Usage of Specific Functions and Classes

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

```
help(np.ones)
```

```
Help on function ones in module mxnet.numpy:

ones(shape, dtype=<class 'numpy.float32'>, order='C', ctx=None)
    Return a new array of given shape and type, filled with ones.
    This function currently only supports storing multi-dimensional data
    in row-major (C-style).

    Parameters
    ----------
    shape : int or tuple of int
        The shape of the empty array.
    dtype : str or numpy.dtype, optional
        An optional value type. Default is numpy.float32. Note that this
        behavior is different from NumPy's ones function where float64
        is the default value, because float32 is considered as the default
        data type in deep learning.
    order : {'C'}, optional, default: 'C'
        How to store multi-dimensional data in memory, currently only row-major
        (C-style) is supported.
    ctx : Context, optional
        An optional device context (default is the current default context).

    Returns
    -------
    out : ndarray
        Array of ones with the given shape, dtype, and ctx.

    Examples
    --------
    >>> np.ones(5)
    array([1., 1., 1., 1., 1.])

    >>> np.ones((5,), dtype=int)
    array([1, 1, 1, 1, 1], dtype=int64)

    >>> np.ones((2, 1))
    array([[1.],
```

```
       [1.]])

>>> s = (2,2)
>>> np.ones(s)
array([[1., 1.],
       [1., 1.]])
```

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:

```
np.ones(4)
```

```
array([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.

Summary

- The official documentation provides plenty of descriptions and examples that are beyond this book.
- We can look up documentation for the usage of an API by calling the `dir` and `help` functions, or ? and ?? in Jupyter notebooks.

Exercises

1. Look up the documentation for any function or class in the deep learning framework. Can you also find the documentation on the official website of the framework?

Discussions[48]

[48] https://discuss.d2l.ai/t/38

3 | Linear Neural Networks

Before we get into the details of deep neural networks, we need to cover the basics of neural network training. In this chapter, we will cover the entire training process, including defining simple neural network architectures, handling data, specifying a loss function, and training the model. In order to make things easier to grasp, we begin with the simplest concepts. Fortunately, classic statistical learning techniques such as linear and softmax regression can be cast as *linear* neural networks. Starting from these classic algorithms, we will introduce you to the basics, providing the basis for more complex techniques in the rest of the book.

3.1 Linear Regression

Regression refers to a set of methods for modeling the relationship between one or more independent variables and a dependent variable. In the natural sciences and social sciences, the purpose of regression is most often to *characterize* the relationship between the inputs and outputs. Machine learning, on the other hand, is most often concerned with *prediction*.

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

3.1.1 Basic Elements of Linear Regression

Linear regression may be both the simplest and most popular among the standard tools to regression. Dating back to the dawn of the 19th century, linear regression flows from a few simple assumptions. First, we assume that the relationship between the independent variables $\mathbf{x}$ and the dependent variable y is linear, i.e., that y can be expressed as a weighted sum of the elements in $\mathbf{x}$, given some noise on the observations. Second, we assume that any noise is well-behaved (following a Gaussian distribution).

To motivate the approach, let us start with 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 actually develop a model for predicting house prices, we would need to get our hands on a dataset consisting of sales for which we know the sale 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 (here the data corresponding to one sale) is called an *example* (or *data point*, *data instance*, *sample*). The thing we are trying to predict (price) is called a *label* (or *target*). The independent variables (age and area) upon which the predictions are based are called *features* (or *covariates*).

Typically, we will use n to denote the number of examples in our dataset. We index the data examples by i, denoting each input as $\mathbf{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}]^\top$ and the corresponding label as $y^{(i)}$.

Linear Model

The linearity assumption just says that the target (price) can be expressed as a weighted sum of the features (area and age):

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

In (3.1.1), w_{area} and w_{age} are called *weights*, and b is called a *bias* (also called an *offset* or *intercept*). The weights determine the influence of each feature on our prediction and the bias just says what value the predicted price should take when all of the features take value 0. Even if we will never see any homes with zero area, or that are precisely zero years old, we still need the bias or else we will limit the expressivity of our model. Strictly speaking, (3.1.1) is an *affine transformation* of input features, which is characterized by a *linear transformation* of features via 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 such that on average, the predictions made according to our model best fit the true prices observed in the data. Models whose output prediction is determined by the affine transformation of input features are *linear models*, where the affine transformation is specified by the chosen weights and bias.

In disciplines where it is common to focus on datasets with just a few features, explicitly expressing models long-form like this is common. In machine learning, we usually work with high-dimensional datasets, so it is more convenient to employ linear algebra notation. When our inputs consist of d features, we express our prediction $\hat{y}$ (in general the "hat" symbol denotes estimates) as

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

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 using a dot product:

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

In (3.1.3), the vector $\mathbf{x}$ corresponds to features of a single data 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 (see Section 2.1.3) 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 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 lowest 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 $\mathbf{y}$ might suffer 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 quality measure for some given model; and (ii) a procedure for updating the model to improve its quality.

Loss Function

Before we start thinking about how to *fit* data with our model, we need to determine a measure of *fitness*. The *loss function* quantifies the distance between the *real* and *predicted* value of the target. The loss will usually be a non-negative number where smaller values are better and perfect predictions incur a loss of 0. The most popular loss function in regression problems 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 will prove notationally convenient, canceling out when we take the derivative of the loss. Since the training dataset is given to us, and thus out of our control, the empirical error is only a function of the model parameters. To make things more concrete, consider the example below where we plot a regression problem for a one-dimensional case as shown in Fig. 3.1.1.

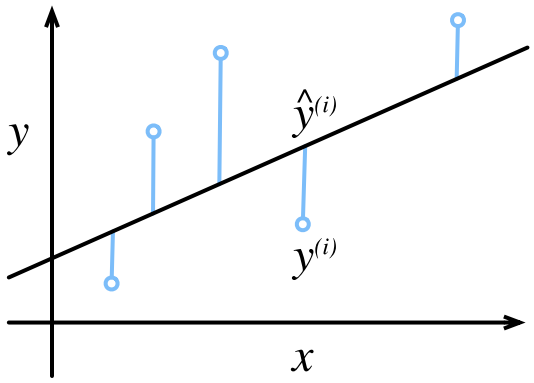


Fig. 3.1.1: Fit data with a linear model.

Note that large differences between estimates $\hat{y}^{(i)}$ and observations $y^{(i)}$ lead to even larger contributions to the loss, due to the quadratic dependence. 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 want to find 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

Linear regression happens to be an unusually simple optimization problem. Unlike most other models that we will encounter in this book, linear regression can be solved analytically by applying a simple formula. To start, we can subsume the bias b into the parameter $\mathbf{w}$ by appending a column to the design matrix consisting of all ones. Then our prediction problem is to minimize $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$. There is 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 the analytic (closed-form) solution:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \tag{3.1.8}$$

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 all of deep learning.

Minibatch Stochastic Gradient Descent

Even in cases where we cannot solve the models analytically, it turns out that we can still train models effectively in practice. Moreover, for many tasks, those difficult-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 any 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. Thus, we will often settle for sampling a random minibatch of examples every time we need to compute the update, a variant called *minibatch stochastic gradient descent*.

In each iteration, we first randomly sample a minibatch $\mathcal{B}$ consisting of a fixed number of training examples. We then compute the derivative (gradient) of the average loss on the minibatch with regard to the model parameters. Finally, we multiply the gradient by a predetermined positive value η and subtract the resulting term from the current parameter values.

We can express the update mathematically as follows (∂ denotes the partial derivative):

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w}, b)} l^{(i)}(\mathbf{w}, b). \tag{3.1.9}$$

To summarize, steps of the algorithm are the following: (i) we initialize the values of the model parameters, typically at random; (ii) we iteratively sample random minibatches from the data, updating the parameters in the direction of the negative gradient. For quadratic losses and affine transformations, we can write this out explicitly as follows:

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right), \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_b l^{(i)}(\mathbf{w}, b) = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right). \end{aligned} \tag{3.1.10}$$

Note that $\mathbf{w}$ and $\mathbf{x}$ are vectors in (3.1.10). Here, the more elegant vector notation makes the math much more readable than expressing things in terms of coefficients, say $w_1, w_2, \ldots, w_d$. The set

cardinality $|\mathcal{B}|$ represents the number of examples in each minibatch (the *batch size*) and η denotes the *learning rate*. We emphasize that the values of the batch size and learning rate are manually pre-specified and not typically learned through model training. These parameters that are tunable but not updated in the training loop are called *hyperparameters*. *Hyperparameter tuning* is the process by which hyperparameters are chosen, and typically requires that we adjust them based on the results of the training loop as assessed on a separate *validation dataset* (or *validation set*).

After training for some predetermined number of iterations (or until some other stopping criteria are 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 because, although the algorithm converges slowly towards the minimizers it cannot achieve it exactly in a finite number of steps.

Linear regression happens to be a learning problem where there is only one minimum over the entire domain. However, for more complicated models, like deep networks, the loss surfaces contain many minima. Fortunately, for reasons that are not yet fully understood, deep learning practitioners seldom struggle to find parameters that minimize the loss *on training sets*. The more formidable task is to find parameters that will achieve low loss on data that we have not seen before, a challenge called *generalization*. We return to these topics throughout the book.

Making Predictions with the Learned Model

Given the learned linear regression model $\hat{\mathbf{w}}^\top \mathbf{x} + \hat{b}$, we can now estimate the price of a new house (not contained in the training data) given its area x_1 and age x_2. Estimating targets given features is commonly called *prediction* or *inference*.

We will try to stick with *prediction* because calling this step *inference*, despite emerging as standard jargon in deep learning, is somewhat of a misnomer. In statistics, *inference* more often denotes estimating parameters based on a dataset. This misuse of terminology is a common source of confusion when deep learning practitioners talk to statisticians.

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.

```
%matplotlib inline
import math
import time
from mxnet import np
from d2l import mxnet as d2l
```

To illustrate why this matters so much, we can consider two methods for adding vectors. To start we instantiate two 10000-dimensional vectors containing all ones. In one method we will loop over the vectors with a Python for-loop. In the other method we will rely on a single call to +.

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

Since we will benchmark the running time frequently in this book, let us define a timer.

```
class Timer:  #@save
    """Record multiple running times."""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        """Start the timer."""
        self.tik = time.time()

    def stop(self):
        """Stop the timer and record the time in a list."""
        self.times.append(time.time() - self.tik)
        return self.times[-1]

    def avg(self):
        """Return the average time."""
        return sum(self.times) / len(self.times)

    def sum(self):
        """Return the sum of time."""
        return sum(self.times)

    def cumsum(self):
        """Return the accumulated time."""
        return np.array(self.times).cumsum().tolist()
```

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

```
c = np.zeros(n)
timer = Timer()
for i in range(n):
    c[i] = a[i] + b[i]
f'{timer.stop():.5f} sec'
```

```
'4.30132 sec'
```

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

```
timer.start()
d = a + b
f'{timer.stop():.5f} sec'
```

```
'0.00037 sec'
```

You probably noticed that 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 and need not write as many calculations ourselves, reducing the potential for errors.

3.1.3 The Normal Distribution and Squared Loss

While you can already get your hands dirty using only the information above, in the following we can more formally motivate the squared loss objective via assumptions about the distribution of noise.

Linear regression was invented by Gauss in 1795, who also discovered the normal distribution (also called the *Gaussian*). It turns out that the connection between the normal distribution and linear regression runs deeper than common parentage. To refresh your memory, the probability density of a normal distribution with mean μ and variance σ^2 (standard deviation σ) is given as

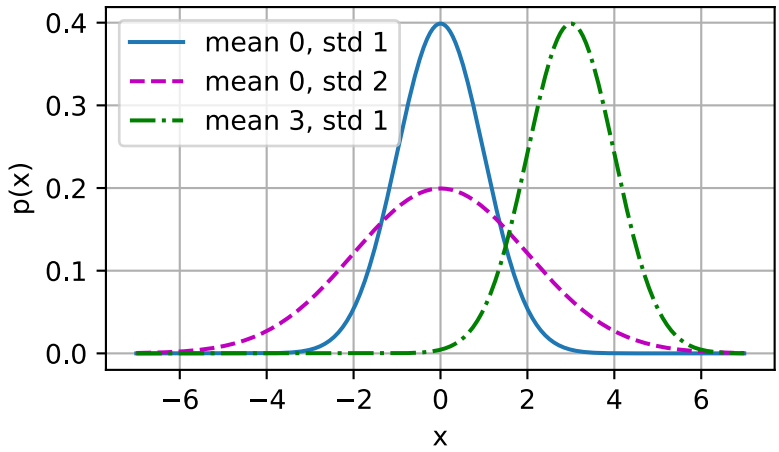
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right). \tag{3.1.11}$$

Below we define a Python 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 / sigma**2 * (x - mu)**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])
```



As we can see, 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 the mean squared error loss function (or simply squared loss) is to formally assume that observations arise from noisy observations, where the noise is normally distributed as follows:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{3.1.12}$$

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

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \tag{3.1.13}$$

Now, 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:

$$P(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}). \tag{3.1.14}$$

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 log 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* $-\log P(\mathbf{y} \mid \mathbf{X})$. Working out the mathematics gives us:

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \tag{3.1.15}$$

Now we just need one more assumption that σ is some fixed constant. Thus we can ignore the first term because it does not depend on $\mathbf{w}$ or b. Now 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. It follows that minimizing the mean squared error is equivalent to maximum likelihood estimation of a linear model under the assumption of additive Gaussian noise.

3.1.4 From Linear Regression to Deep Networks

So far we only talked about linear models. While neural networks cover a much richer family of models, we can begin thinking of the linear model as a neural network by expressing it in the language of neural networks. To begin, let us start by rewriting things in a "layer" notation.

Neural Network Diagram

Deep learning practitioners like to draw diagrams to visualize what is happening in their models. In Fig. 3.1.2, we depict our linear regression model as a neural network. Note that these diagrams highlight the connectivity pattern such as how each input is connected to the output, but not the values taken by the weights or biases.

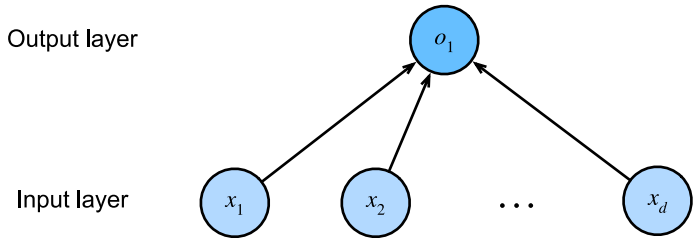


Fig. 3.1.2: Linear regression is a single-layer neural network.

For the neural network shown in Fig. 3.1.2, the inputs are $x_1, \ldots, x_d$, so the *number of inputs* (or *feature dimensionality*) in the input layer is d. The output of the network in Fig. 3.1.2 is o_1, so the

number of outputs in the output layer is 1. Note that the input values are all *given* and there is just a single *computed* neuron. Focusing on where computation takes place, conventionally we do not consider the input layer when counting layers. That is to say, the *number of layers* for the neural network in Fig. 3.1.2 is 1. We can think of linear regression models as neural networks consisting of just a single artificial neuron, or as single-layer neural networks.

Since for linear regression, every input is connected to every output (in this case there is only one output), we can regard this transformation (the output layer in Fig. 3.1.2) as a *fully-connected layer* or *dense layer*. We will talk a lot more about networks composed of such layers in the next chapter.

Biology

Since linear regression (invented in 1795) predates computational neuroscience, it might seem anachronistic to describe linear regression as a neural network. To see why linear models were a natural place to begin when the cyberneticists/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*.

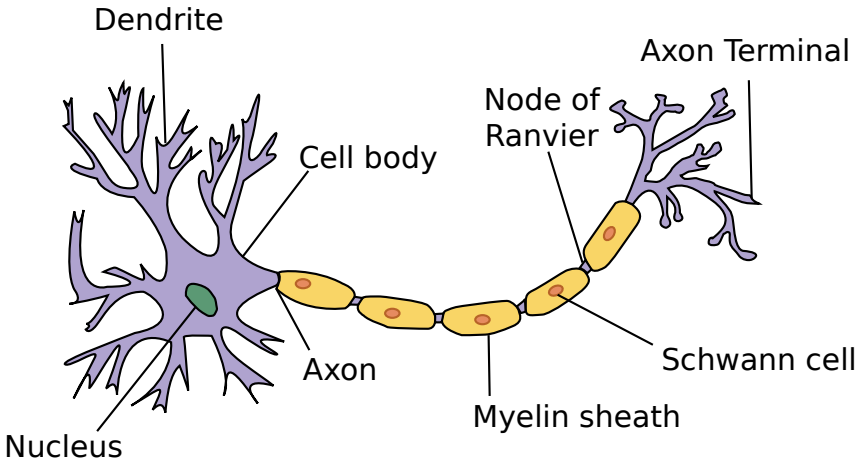


Fig. 3.1.3: The real neuron.

Information x_i arriving from other neurons (or environmental sensors such as the retina) 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 $y = \sum_i x_i w_i + b$, and this information is then sent for further processing in the axon y, typically after some nonlinear processing via $\sigma(y)$. From there it either reaches its destination (e.g., a muscle) or is fed into another neuron via its dendrites.

Certainly, the high-level idea that many such units could be cobbled together with the right connectivity and right learning algorithm, to produce far more interesting and complex behavior than any one neuron alone could express owes to our study of real biological neural systems.

At the same time, most research in deep learning today draws little direct inspiration in neuroscience. We invoke Stuart Russell and Peter Norvig who, in their classic AI text book *Artificial Intelligence: A Modern Approach* (Russell & Norvig, 2016), 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, statistics, and computer science.

Summary

- Key ingredients in a machine learning model are training data, a loss function, an optimization algorithm, and quite obviously, the model itself.
- Vectorizing makes everything better (mostly math) and faster (mostly code).
- Minimizing an objective function and performing maximum likelihood estimation can mean the same thing.
- Linear regression models are neural networks, too.

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 a analytic solution for the optimal value of b.
    2. How does this problem and its solution relate to the normal distribution?
2. Derive the analytic solution to the optimization problem for linear regression with squared error. To keep things simple, you can omit the bias b from the problem (we can do this in principled fashion by adding one column to $\mathbf{X}$ consisting of all ones).
    1. Write out the optimization problem in matrix and vector notation (treat all the data as a single matrix, and all the target values as a single vector).
    2. Compute the gradient of the loss with respect to w.
    3. Find the analytic solution by setting the gradient equal to zero and solving the matrix equation.
    4. When might this be better than using stochastic gradient descent? When might this method break?
3. 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 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?

Discussions[49]

[49] https://discuss.d2l.ai/t/40

3.2 Linear Regression Implementation from Scratch

Now that you understand the key ideas behind linear regression, we can begin to work through a hands-on implementation in code. In this section, we will implement the entire method from scratch, including the data pipeline, the model, the loss function, and the minibatch stochastic gradient descent optimizer. 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 comes 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 auto differentiation. Afterwards, we will introduce a more concise implementation, taking advantage of bells and whistles of deep learning frameworks.

```
%matplotlib inline
import random
from mxnet import autograd, np, npx
from d2l import mxnet as d2l

npx.set_np()
```

3.2.1 Generating the Dataset

To keep things simple, we will construct an artificial dataset according to a linear model with additive noise. Our task will be to recover this model's parameters using the finite set of examples contained in our dataset. We will keep the data low-dimensional so we can visualize it easily. In the following code snippet, we generate a dataset containing 1000 examples, each consisting of 2 features sampled from a standard normal distribution. Thus our synthetic dataset will be a matrix $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$.

The true parameters generating our dataset will be $\mathbf{w} = [2, -3.4]^\top$ and $b = 4.2$, and our synthetic labels will be assigned according to the following linear model with the noise term ϵ:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon. \tag{3.2.1}$$

You could think of ϵ as capturing potential measurement errors on the features and labels. We will assume that the standard assumptions hold and thus that ϵ obeys a normal distribution with mean of 0. To make our problem easy, we will set its standard deviation to 0.01. The following code generates our synthetic dataset.

```
def synthetic_data(w, b, num_examples):  #@save
    """Generate y = Xw + b + noise."""
    X = np.random.normal(0, 1, (num_examples, len(w)))
    y = np.dot(X, w) + b
    y += np.random.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1))
```

```
true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

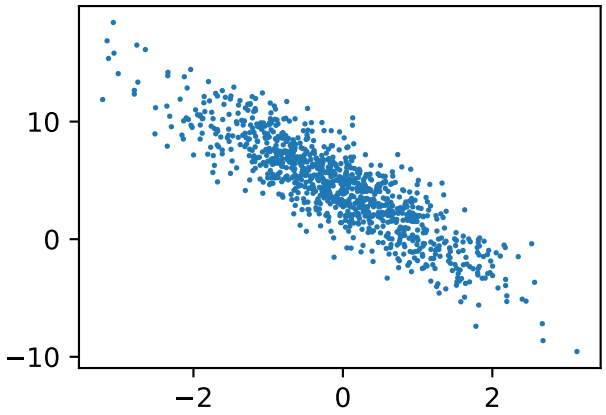
Note that each row in `features` consists of a 2-dimensional data example and that each row in `labels` consists of a 1-dimensional label value (a scalar).

```
print('features:', features[0], '\nlabel:', labels[0])
```

```
features: [2.2122064 1.1630787]
label: [4.662078]
```

By generating a scatter plot using the second feature `features[:, 1]` and `labels`, we can clearly observe the linear correlation between the two.

```
d2l.set_figsize()
# The semicolon is for displaying the plot only
d2l.plt.scatter(features[:, (1)].asnumpy(), labels.asnumpy(), 1);
```



3.2.2 Reading the Dataset

Recall that training models consists of making multiple passes over the dataset, grabbing one minibatch of examples at a time, and using them to update our model. Since this process is so fundamental to training machine learning algorithms, it is worth defining a utility function to shuffle the dataset and access it in minibatches.

In the following code, we define the `data_iter` function to demonstrate one possible implementation of this functionality. The function takes a batch size, a matrix of features, and a vector of labels, yielding minibatches of the size `batch_size`. Each minibatch consists of a tuple of features and labels.

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = np.array(indices[i:min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

In general, note that we want to use reasonably sized minibatches to take advantage of the GPU hardware, which excels at parallelizing operations. Because each example can be fed through our models in parallel and the gradient of the loss function for each example can also be taken in

parallel, GPUs allow us to process hundreds of examples in scarcely more time than it might take to process just a single example.

To build some intuition, let us read and print the first small batch of data examples. The shape of the features in each minibatch tells us both the minibatch size and the number of input features. Likewise, our minibatch of labels will have a shape given by `batch_size`.

```
batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break
```

```
[[-0.93316925  1.5430632 ]
 [ 0.1737154  -1.3096327 ]
 [-0.17182696  0.71263075]
 [-1.0310302  -1.0057124 ]
 [ 0.8591078   0.66443396]
 [-0.35391203  1.2594353 ]
 [-0.5836186  -0.07530449]
 [-0.04391905 -1.0315224 ]
 [ 0.8468736  -0.55909204]
 [-0.9354301  -0.9410188 ]] 
 [[-2.9196413 ]
 [ 9.009764  ]
 [ 1.4362991 ]
 [ 5.54579   ]
 [ 3.6709807 ]
 [-0.78678733]
 [ 3.2921002 ]
 [ 7.6282578 ]
 [ 7.7833023 ]
 [ 5.5244083 ]]
```

As we run the iteration, we obtain distinct minibatches successively 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 in trouble on 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 both data stored in files and data fed via data streams.

3.2.3 Initializing Model Parameters

Before we can begin optimizing our model's parameters by minibatch stochastic gradient descent, we need to have some parameters in the first place. In the following code, we initialize weights by sampling random numbers from a normal distribution with mean 0 and a standard deviation of 0.01, and setting the bias to 0.

```
w = np.random.normal(0, 0.01, (2, 1))
b = np.zeros(1)
w.attach_grad()
b.attach_grad()
```

After initializing our parameters, our next task is to update them until they fit our data sufficiently well. Each update requires taking the gradient of our loss function with respect to the parameters. Given this gradient, we can update each parameter in the direction that may reduce the loss.

Since nobody wants to compute gradients explicitly (this is tedious and error prone), we use automatic differentiation, as introduced in Section 2.5, to compute the gradient.

3.2.4 Defining the Model

Next, we must define our model, relating its inputs and parameters to its outputs. Recall that to calculate the output of the linear model, we simply take the matrix-vector dot product of the input features $\mathbf{X}$ and the model weights $\mathbf{w}$, and add the offset b to each example. Note that below $\mathbf{Xw}$ is a vector and b is a scalar. Recall the broadcasting mechanism as described in Section 2.1.3. When we add a vector and a scalar, the scalar is added to each component of the vector.

```
def linreg(X, w, b):  #@save
    """The linear regression model."""
    return np.dot(X, w) + b
```

3.2.5 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 will use the squared loss function as described in Section 3.1. 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 function will also have the same shape as `y_hat`.

```
def squared_loss(y_hat, y):  #@save
    """Squared loss."""
    return (y_hat - y.reshape(y_hat.shape))**2 / 2
```

3.2.6 Defining the Optimization Algorithm

As we discussed in Section 3.1, linear regression has a closed-form solution. However, this is not a book about linear regression: it is a book about deep learning. Since none of the other models that this book introduces can be solved analytically, we will take this opportunity to introduce your first working example of minibatch stochastic gradient descent.

At each step, using one minibatch randomly drawn from our dataset, we will estimate the gradient of the loss with respect to our parameters. Next, we will update our parameters in the direction that may reduce the loss. The following code applies the minibatch stochastic gradient descent update, given a set of parameters, a learning rate, and a batch size. The size of the update step is determined by the learning rate `lr`. Because our loss is calculated as a sum over the minibatch of examples, we normalize our step size by the batch size (`batch_size`), so that the magnitude of a typical step size does not depend heavily on our choice of the batch size.

```
def sgd(params, lr, batch_size):  #@save
    """Minibatch stochastic gradient descent."""
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

3.2.7 Training

Now that we have all of the parts in place, we are ready to implement the main training loop. It is crucial that you understand this code because you will see nearly identical training loops over and over again throughout your career in deep learning.

In each iteration, we will grab a minibatch of training examples, and pass them through our model to obtain a set of predictions. After calculating the loss, we initiate the backwards pass through the network, storing the gradients with respect to each parameter. Finally, we will call the optimization algorithm `sgd` 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 $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

In each *epoch*, we will iterate through the entire dataset (using the `data_iter` function) once passing through every example in the training dataset (assuming that the number of examples is divisible by the batch size). The number of epochs `num_epochs` and the learning rate `lr` are both hyperparameters, which we set here to 3 and 0.03, respectively. Unfortunately, setting hyperparameters is tricky and requires some adjustment by trial and error. We elide these details for now but revise them later in Chapter 11.

```
lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss
```

```
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        with autograd.record():
            l = loss(net(X, w, b), y)  # Minibatch loss in `X` and `y`
        # Because `l` has a shape (`batch_size`, 1) and is not a scalar
        # variable, the elements in `l` are added together to obtain a new
        # variable, on which gradients with respect to [`w`, `b`] are computed
        l.backward()
        sgd([w, b], lr, batch_size)  # Update parameters using their gradient
    train_l = loss(net(features, w, b), labels)
    print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')
```

```
epoch 1, loss 0.025025
epoch 2, loss 0.000093
epoch 3, loss 0.000051
```

In this case, 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.

```
print(f'error in estimating w: {true_w - w.reshape(true_w.shape)}')
print(f'error in estimating b: {true_b - b}')
```

```
error in estimating w: [0.00039625 0.00020432]
error in estimating b: [0.00054502]
```

Note that we should not take it for granted that we are able to recover the parameters perfectly. However, in machine learning, we are typically less concerned with recovering true underlying parameters, and more concerned with parameters that lead to highly accurate prediction. 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.

Summary

- We saw how a deep network can be implemented and optimized from scratch, using just tensors and auto differentiation, without any need for defining layers or fancy optimizers.
- This section only scratches the surface of what is possible. In the following sections, we will describe additional models based on the concepts that we have just introduced and learn how to implement them more concisely.

Exercises

1. What would happen if we were to initialize the weights to zero. Would the algorithm still work?
2. Assume that you are Georg Simon Ohm[50] trying to come up with a model between voltage and current. Can you use auto differentiation to learn the parameters of your model?
3. Can you use Planck's Law[51] to determine the temperature of an object using spectral energy density?
4. What are the problems you might encounter if you wanted to compute the second derivatives? How would you fix them?
5. Why is the `reshape` function needed in the `squared_loss` function?
6. Experiment using different learning rates to find out how fast the loss function value drops.
7. If the number of examples cannot be divided by the batch size, what happens to the `data_iter` function's behavior?

Discussions[52]

[50] https://en.wikipedia.org/wiki/Georg_Ohm
[51] https://en.wikipedia.org/wiki/Planck%27s_law
[52] https://discuss.d2l.ai/t/42

3.3 Concise Implementation of Linear Regression

Broad and intense interest in deep learning for the past several years has inspired companies, academics, and hobbyists to develop a variety of mature open source frameworks for automating the repetitive work of implementing gradient-based learning algorithms. In Section 3.2, we relied only on (i) tensors for data storage and linear algebra; and (ii) auto 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 3.2 concisely by using high-level APIs of deep learning frameworks.

3.3.1 Generating the Dataset

To start, we will generate the same dataset as in Section 3.2.

```
from mxnet import autograd, gluon, np, npx
from d2l import mxnet as d2l

npx.set_np()
```

```
true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

3.3.2 Reading the Dataset

Rather than rolling our own iterator, we can call upon the existing API in a framework to read data. We pass in `features` and `labels` as arguments and specify `batch_size` when instantiating a data iterator object. Besides, the boolean value `is_train` indicates whether or not we want the data iterator object to shuffle the data on each epoch (pass through the dataset).

```
def load_array(data_arrays, batch_size, is_train=True):  #@save
    """Construct a Gluon data iterator."""
    dataset = gluon.data.ArrayDataset(*data_arrays)
    return gluon.data.DataLoader(dataset, batch_size, shuffle=is_train)
```

```
batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

Now we can use `data_iter` in much the same way as we called the `data_iter` function in Section 3.2. To verify that it is working, we can read and print the first minibatch of examples. Comparing with Section 3.2, here we use `iter` to construct a Python iterator and use `next` to obtain the first item from the iterator.

```
next(iter(data_iter))
```

```
[array([[ 0.67848086, -0.7045922 ],
        [-0.9669159 , -0.8711447 ],
        [-1.0352775 ,  1.6948094 ],
        [ 0.8515685 ,  0.12359928],
        [-0.6308071 ,  0.53733146],
        [ 0.6862904 ,  1.07628   ],
        [ 0.1130666 , -0.659559  ],
        [ 0.308205  ,  0.9416017 ],
        [-0.34304565,  0.9344402 ],
        [-1.5367762 ,  1.0635569 ]]),
 array([[ 7.961054 ],
        [ 5.2282877],
        [-3.6350167],
        [ 5.4853864],
        [ 1.0948168],
        [ 1.9102741],
        [ 6.6553063],
        [ 1.6211131],
        [ 0.3322031],
        [-2.4930956]])]
```

3.3.3 Defining the Model

When we implemented linear regression from scratch in Section 3.2, 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 for 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 every time you needed a blog you spent a month reinventing the wheel.

For standard operations, we can use a framework's predefined layers, which allow us to focus especially on the layers used to construct the model rather than having to focus on the implementation. We will first define a model variable `net`, which will refer to an instance of the `Sequential` class. The `Sequential` class defines a container for several layers that will be chained together. Given input data, a `Sequential` instance passes it through the first layer, in turn passing the output as the second layer's input and so forth. In the following example, our model consists of only one layer, so we do not really need `Sequential`. But since nearly all of our future models will involve multiple layers, we will use it anyway just to familiarize you with the most standard workflow.

Recall the architecture of a single-layer network as shown in Fig. 3.1.2. The layer is said to be *fully-connected* because each of its inputs is connected to each of its outputs by means of a matrix-vector multiplication.

In Gluon, the fully-connected layer is defined in the `Dense` class. Since we only want to generate a single scalar output, we set that number to 1.

It is worth noting that, for convenience, Gluon does not require us to specify the input shape for each layer. So here, we do not need to tell Gluon how many inputs go into this linear layer. When we first try to pass data through our model, e.g., when we execute `net(X)` later, Gluon will automatically infer the number of inputs to each layer. We will describe how this works in more detail later.

```
# `nn` is an abbreviation for neural networks
from mxnet.gluon import nn

net = nn.Sequential()
net.add(nn.Dense(1))
```

3.3.4 Initializing Model Parameters

Before using `net`, we need to initialize the model parameters, such as the weights and bias in the linear regression model. Deep learning frameworks often have a predefined way to initialize the parameters. Here we specify that each weight parameter should be randomly sampled from a normal distribution with mean 0 and standard deviation 0.01. The bias parameter will be initialized to zero.

We will import the `initializer` module from MXNet. This module provides various methods for model parameter initialization. Gluon makes `init` available as a shortcut (abbreviation) to access the `initializer` package. We only specify how to initialize the weight by calling `init.Normal(sigma=0.01)`. Bias parameters are initialized to zero by default.

```
from mxnet import init

net.initialize(init.Normal(sigma=0.01))
```

The code above may look straightforward but you should note that something strange is happening here. We are initializing parameters for a network even though Gluon does not yet know how many dimensions the input will have! It might be 2 as in our example or it might be 2000. Gluon lets us get away with this because behind the scene, the initialization is actually *deferred*. The real initialization will take place only when we for the first time attempt to pass data through the network. Just be careful to remember that since the parameters have not been initialized yet, we cannot access or manipulate them.

3.3.5 Defining the Loss Function

In Gluon, the `loss` module defines various loss functions. In this example, we will use the Gluon implementation of squared loss (`L2Loss`).

```
loss = gluon.loss.L2Loss()
```

3.3.6 Defining the Optimization Algorithm

Minibatch stochastic gradient descent is a standard tool for optimizing neural networks and thus Gluon supports it alongside a number of variations on this algorithm through its `Trainer` class. When we instantiate `Trainer`, we will specify the parameters to optimize over (obtainable from our model `net` via `net.collect_params()`), the optimization algorithm we wish to use (`sgd`), and a dictionary of hyperparameters required by our optimization algorithm. Minibatch stochastic gradient descent just requires that we set the value `learning_rate`, which is set to 0.03 here.

```
from mxnet import gluon

trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

3.3.7 Training

You might have noticed that expressing our model through high-level APIs of a deep learning framework requires comparatively few lines of code. We did not have to individually allocate parameters, define our loss function, or implement minibatch stochastic gradient descent. Once we start working with much more complex models, advantages of high-level APIs will grow considerably. However, once we have all the basic pieces in place, the training loop itself is strikingly similar to what we did when implementing everything from scratch.

To refresh your memory: for some number of epochs, we will make a complete pass over the dataset (`train_data`), iteratively grabbing one minibatch of inputs and the corresponding ground-truth labels. For each minibatch, we go through the following ritual:

- Generate predictions by calling `net(X)` and calculate the loss `l` (the forward propagation).
- Calculate gradients by running the backpropagation.
- Update the model parameters by invoking our optimizer.

For good measure, we compute the loss after each epoch and print it to monitor progress.

```
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        with autograd.record():
            l = loss(net(X), y)
        l.backward()
        trainer.step(batch_size)
    l = loss(net(features), labels)
    print(f'epoch {epoch + 1}, loss {l.mean().asnumpy():f}')
```

```
epoch 1, loss 0.024892
epoch 2, loss 0.000090
epoch 3, loss 0.000051
```

Below, we compare the model parameters learned by training on finite data and the actual parameters that generated our dataset. To access parameters, we first access the layer that we need from `net` and then access that layer's weights and bias. As in our from-scratch implementation, note that our estimated parameters are close to their ground-truth counterparts.

```
w = net[0].weight.data()
print(f'error in estimating w: {true_w - w.reshape(true_w.shape)}')
b = net[0].bias.data()
print(f'error in estimating b: {true_b - b}')
```

```
error in estimating w: [ 8.7058544e-04 -3.8385391e-05]
error in estimating b: [0.00039387]
```

Summary

- Using Gluon, we can implement models much more concisely.
- In Gluon, the `data` module provides tools for data processing, the `nn` module defines a large number of neural network layers, and the `loss` module defines many common loss functions.
- MXNet's module `initializer` provides various methods for model parameter initialization.
- Dimensionality and storage are automatically inferred, but be careful not to attempt to access parameters before they have been initialized.

Exercises

1. If we replace `l = loss(output, y)` with `l = loss(output, y).mean()`, we need to change `trainer.step(batch_size)` to `trainer.step(1)` for the code to behave identically. Why?
2. Review the MXNet documentation to see what loss functions and initialization methods are provided in the modules `gluon.loss` and `init`. Replace the loss by Huber's loss.
3. How do you access the gradient of `dense.weight`?

Discussions[53]

3.4 Softmax Regression

In Section 3.1, we introduced linear regression, working through implementations from scratch in Section 3.2 and again using high-level APIs of a deep learning framework in Section 3.3 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.

In practice, we are more often interested in *classification*: asking not "how much" but "which one":

- 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?

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.

[53] https://discuss.d2l.ai/t/44

3.4.1 Classification Problem

To get our feet wet, let us start off 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 us 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 problem as regression and keep the labels in this format.

But 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 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":

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \tag{3.4.1}$$

3.4.2 Network Architecture

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. Each output will correspond to its own affine function. In our case, since we have 4 features and 3 possible output categories, we will need 12 scalars to represent the weights (w with subscripts), and 3 scalars to represent the biases (b with subscripts). We compute these three *logits*, o_1, o_2, and o_3, for each input:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned} \tag{3.4.2}$$

We can depict this calculation with the neural network diagram shown in Fig. 3.4.1. Just as in linear regression, softmax regression is also a single-layer neural network. And since the calculation of each output, o_1, o_2, and o_3, depends on all inputs, x_1, x_2, x_3, and x_4, the output layer of softmax regression can also be described as fully-connected layer.

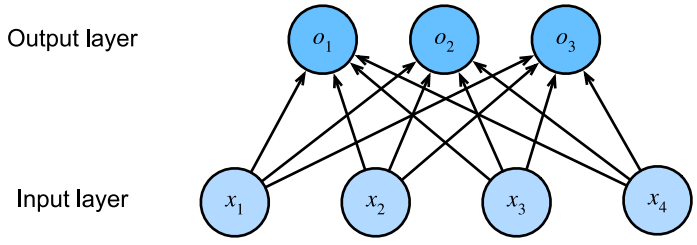


Fig. 3.4.1: Softmax regression is a single-layer neural network.

To express the model more compactly, we can use linear algebra notation. In vector form, we arrive at $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$, a form better suited both for mathematics, and for writing code. Note that

we have gathered all of our weights into a 3×4 matrix and that for features of a given data example $\mathbf{x}$, our outputs are given by a matrix-vector product of our weights by our input features plus our biases $\mathbf{b}$.

3.4.3 Parameterization Cost of Fully-Connected Layers

As we will see in subsequent chapters, fully-connected layers are ubiquitous in deep learning. However, as the name suggests, fully-connected layers are *fully* connected with potentially many learnable parameters. Specifically, for any fully-connected layer with d inputs and q outputs, the parameterization cost is $\mathcal{O}(dq)$, which can be prohibitively high in practice. Fortunately, this cost of transforming d inputs into q outputs can be reduced to $\mathcal{O}(\frac{dq}{n})$, where the hyperparameter n can be flexibly specified by us to balance between parameter saving and model effectiveness in real-world applications (Zhang et al., 2021).

3.4.4 Softmax Operation

The main approach that we are going to take here is to interpret the outputs of our model as probabilities. We will optimize our parameters to produce probabilities that maximize the likelihood of the observed data. Then, to generate predictions, we will set a threshold, for example, choosing the label with the maximum predicted probabilities.

Put formally, we would like any output $\hat{y}_j$ to be interpreted as the probability that a given item belongs to class j. Then we can choose the class with the largest output value as our prediction $\operatorname{argmax}_j y_j$. For example, if $\hat{y}_1$, $\hat{y}_2$, and $\hat{y}_3$ are 0.1, 0.8, and 0.1, respectively, then we predict category 2, which (in our example) represents "chicken".

You might be tempted to suggest that we interpret the logits o directly as our outputs of interest. However, there are some problems with directly interpreting the output of the linear layer as a probability. On one hand, nothing constrains these numbers to sum to 1. On the other hand, depending on the inputs, they can take negative values. These violate basic axioms of probability presented in Section 2.6

To interpret our outputs as probabilities, we must guarantee that (even on new data), they will be nonnegative and sum up to 1. Moreover, we need a training objective that encourages the model to estimate faithfully probabilities. Of all instances when a classifier outputs 0.5, we hope that half of those examples will actually belong to the predicted class. This is a property called *calibration*.

The *softmax function*, invented in 1959 by the social scientist R. Duncan Luce in the context of *choice models*, does precisely this. To transform our logits such that they become nonnegative and sum to 1, while requiring that the model remains differentiable, we first exponentiate each logit (ensuring non-negativity) and then divide by their sum (ensuring that they sum to 1):

$$\hat{\mathbf{y}} = \operatorname{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}. \tag{3.4.3}$$

It is easy to see $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ with $0 \leq \hat{y}_j \leq 1$ for all j. Thus, $\hat{\mathbf{y}}$ is a proper probability distribution whose element values can be interpreted accordingly. Note that the softmax operation does not change the ordering among the logits $\mathbf{o}$, which are simply the pre-softmax values that determine the probabilities assigned to each class. Therefore, during prediction we can still pick out the most likely class by

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j. \tag{3.4.4}$$

Although softmax is a nonlinear function, the outputs of softmax regression are still *determined* by an affine transformation of input features; thus, softmax regression is a linear model.

3.4.5 Vectorization for Minibatches

To improve computational efficiency and take advantage of GPUs, we typically carry out vector calculations for minibatches of data. Assume that we are given a minibatch $\mathbf{X}$ of examples with feature dimensionality (number of inputs) d and batch size n. Moreover, assume that we have q categories in the output. Then the minibatch features $\mathbf{X}$ are in $\mathbb{R}^{n \times d}$, weights $\mathbf{W} \in \mathbb{R}^{d \times q}$, and the bias satisfies $\mathbf{b} \in \mathbb{R}^{1 \times q}$.

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \mathrm{softmax}(\mathbf{O}). \end{aligned} \tag{3.4.5}$$

This accelerates the dominant operation into a matrix-matrix product $\mathbf{X}\mathbf{W}$ vs. the matrix-vector products we would be executing if we processed one example at a time. 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. Triggering broadcasting during the summation $\mathbf{X}\mathbf{W} + \mathbf{b}$ in (3.4.5), both the minibatch logits $\mathbf{O}$ and output probabilities $\hat{\mathbf{Y}}$ are $n \times q$ matrices.

3.4.6 Loss Function

Next, we need a loss function to measure the quality of our predicted probabilities. We will rely on maximum likelihood estimation, the very same concept that we encountered when providing a probabilistic justification for the mean squared error objective in linear regression (Section 3.1.3).

Log-Likelihood

The softmax function gives us a vector $\hat{\mathbf{y}}$, which we can interpret as estimated conditional probabilities of each class given any input $\mathbf{x}$, e.g., $\hat{y}_1 = P(y = \text{cat} \mid \mathbf{x})$. Suppose that the entire dataset $\{\mathbf{X}, \mathbf{Y}\}$ has n examples, where the example indexed by i consists of a feature vector $\mathbf{x}^{(i)}$ and a one-hot label vector $\mathbf{y}^{(i)}$. We can compare the estimates with reality by checking how probable the actual classes are according to our model, given the features:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}). \tag{3.4.6}$$

According to maximum likelihood estimation, we maximize $P(\mathbf{Y} \mid \mathbf{X})$, which is equivalent to minimizing the negative log-likelihood:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \tag{3.4.7}$$

where for any pair of label $\mathbf{y}$ and model prediction $\hat{\mathbf{y}}$ over q classes, the loss function l is

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j. \tag{3.4.8}$$

For reasons explained later on, the loss function in (3.4.8) 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. Since all $\hat{y}_j$ are predicted probabilities, their logarithm is never larger than 0. Consequently, the loss function cannot be minimized any further if we correctly predict the actual label with *certainty*, i.e., if the predicted probability $P(\mathbf{y} \mid \mathbf{x}) = 1$ for the actual label $\mathbf{y}$. Note that this is often impossible. For example, there might be label noise in the dataset (some examples may be mislabeled). It may also not be possible when the input features are not sufficiently informative to classify every example perfectly.

Softmax and Derivatives

Since the softmax and the corresponding loss are so common, it is worth understanding a bit better how it is computed. Plugging (3.4.3) into the definition of the loss in (3.4.8) and using the definition of the softmax we obtain:

$$\begin{aligned} l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} \\ &= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \\ &= \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j. \end{aligned} \tag{3.4.9}$$

To understand a bit better what is going on, consider the derivative with respect to any logit o_j. We get

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} - y_j = \mathrm{softmax}(\mathbf{o})_j - y_j. \tag{3.4.10}$$

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 coincidence. In any exponential family (see the online appendix on distributions[54]) model, the gradients of the log-likelihood are given by precisely this term. This fact makes computing gradients easy in practice.

Cross-Entropy Loss

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 (3.4.8) still works out fine, 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. If you wish to understand more details of information theory, you may further refer to the online appendix on information theory[55].

[54] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/distributions.html
[55] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/information-theory.html

3.4.7 Information Theory Basics

Information theory deals with the problem of encoding, decoding, transmitting, and manipulating information (also known as data) in as concise form as possible.

Entropy

The central idea in information theory is to quantify the information content in data. This quantity places a hard limit on our ability to compress the data. In information theory, this quantity is called the *entropy* of a distribution P, and it is captured by the following equation:

$$H[P] = \sum_j -P(j) \log P(j). \tag{3.4.11}$$

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. 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 they 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 we assigned an event 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 (3.4.11) 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 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 were actually generated according to probabilities P. 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.

3.4.8 Model Prediction and Evaluation

After training the softmax regression model, given any example features, we can predict the probability of each output class. Normally, we use the class with the highest predicted probability as the output class. The prediction is correct if it is consistent with the actual class (label). In the next part of the experiment, we will use *accuracy* to evaluate the model's performance. This is equal to the ratio between the number of correct predictions and the total number of predictions.

Summary

- The softmax operation takes a vector and maps it into probabilities.
- Softmax regression applies to classification problems. It uses the probability distribution of the output class in the softmax operation.
- Cross-entropy is a good measure of the difference between two probability distributions. It measures the number of bits needed to encode the data given our model.

Exercises

1. We can explore the connection between exponential families and the softmax in some more depth.
    1. Compute the second derivative of the cross-entropy loss $l(\mathbf{y}, \hat{\mathbf{y}})$ for the softmax.
    2. Compute the variance of the distribution given by $\text{softmax}(\mathbf{o})$ 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. Softmax is a misnomer for the mapping introduced above (but everyone in deep learning uses it). The real softmax is defined as $\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$.
    1. Prove that $\text{RealSoftMax}(a, b) > \max(a, b)$.
    2. Prove that this holds for $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b)$, provided that $\lambda > 0$.
    3. Show that for $\lambda \to \infty$ we have $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$.
    4. What does the soft-min look like?
    5. Extend this to more than two numbers.

Discussions[56]

[56] https://discuss.d2l.ai/t/46

3.5 The Image Classification Dataset

One of the widely used dataset for image classification is the MNIST dataset (LeCun et al., 1998). 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 stronger models and weaker ones. Today, MNIST serves as more of sanity checks than as a benchmark. To up the ante just a bit, we will focus our discussion in the coming sections on the qualitatively similar, but comparatively complex Fashion-MNIST dataset (Xiao et al., 2017), which was released in 2017.

```
%matplotlib inline
import sys
from mxnet import gluon
from d2l import mxnet as d2l

d2l.use_svg_display()
```

3.5.1 Reading the Dataset

We can download and read the Fashion-MNIST dataset into memory via the build-in functions in the framework.

```
mnist_train = gluon.data.vision.FashionMNIST(train=True)
mnist_test = gluon.data.vision.FashionMNIST(train=False)
```

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* (or *test set*) is used for evaluating model performance and not for training. Consequently the training set and the test set contain 60000 and 10000 images, respectively.

```
len(mnist_train), len(mnist_test)
```

```
(60000, 10000)
```

The height and width of each input image are both 28 pixels. Note that the dataset consists of grayscale images, whose number of channels is 1. For brevity, throughout this book we store the shape of any image with height h width w pixels as $h \times w$ or (h, w).

```
mnist_train[0][0].shape
```

```
(28, 28, 1)
```

The images in Fashion-MNIST are associated with the following categories: t-shirt, trousers, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot. The following function converts between numeric label indices and their names in text.

```
def get_fashion_mnist_labels(labels):  #@save
    """Return text labels for the Fashion-MNIST dataset."""
    text_labels = [
```

(continues on next page)

(continued from previous page)

```
        't-shirt', 'trouser', 'pullover', 'dress', 'coat', 'sandal', 'shirt',
        'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]
```

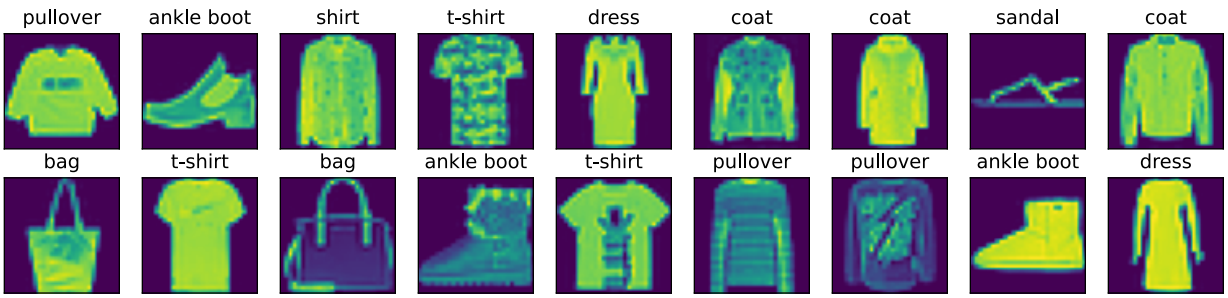
We can now create a function to visualize these examples.

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """Plot a list of images."""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        ax.imshow(img.asnumpy())
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

Here are the images and their corresponding labels (in text) for the first few examples in the training dataset.

```
X, y = mnist_train[:18]

print(X.shape)
show_images(X.squeeze(axis=-1), 2, 9, titles=get_fashion_mnist_labels(y));
```

```
(18, 28, 28, 1)
```



3.5.2 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 loader reads a minibatch of data with size `batch_size` each time. We also randomly shuffle the examples for the training data iterator.

```
batch_size = 256

def get_dataloader_workers():  #@save
    """Use 4 processes to read the data except for Windows."""
```

(continues on next page)

(continued from previous page)

```
    return 0 if sys.platform.startswith('win') else 4

# `ToTensor` converts the image data from uint8 to 32-bit floating point. It
# divides all numbers by 255 so that all pixel values are between 0 and 1
transformer = gluon.data.vision.transforms.ToTensor()
train_iter = gluon.data.DataLoader(mnist_train.transform_first(transformer),
                                   batch_size, shuffle=True,
                                   num_workers=get_dataloader_workers())
```

Let us look at the time it takes to read the training data.

```
timer = d2l.Timer()
for X, y in train_iter:
    continue
f'{timer.stop():.2f} sec'
```

```
'1.89 sec'
```

3.5.3 Putting All Things Together

Now we define the `load_data_fashion_mnist` function that obtains and reads the Fashion-MNIST dataset. It returns the data iterators for both the training set and validation set. In addition, it accepts an optional argument to resize images to another shape.

```
def load_data_fashion_mnist(batch_size, resize=None):  #@save
    """Download the Fashion-MNIST dataset and then load it into memory."""
    dataset = gluon.data.vision
    trans = [dataset.transforms.ToTensor()]
    if resize:
        trans.insert(0, dataset.transforms.Resize(resize))
    trans = dataset.transforms.Compose(trans)
    mnist_train = dataset.FashionMNIST(train=True).transform_first(trans)
    mnist_test = dataset.FashionMNIST(train=False).transform_first(trans)
    return (gluon.data.DataLoader(mnist_train, batch_size, shuffle=True,
                                  num_workers=get_dataloader_workers()),
            gluon.data.DataLoader(mnist_test, batch_size, shuffle=False,
                                  num_workers=get_dataloader_workers()))
```

Below we test the image resizing feature of the `load_data_fashion_mnist` function by specifying the `resize` argument.

```
train_iter, test_iter = load_data_fashion_mnist(32, resize=64)
for X, y in train_iter:
    print(X.shape, X.dtype, y.shape, y.dtype)
    break
```

```
(32, 1, 64, 64) <class 'numpy.float32'> (32,) <class 'numpy.int32'>
```

We are now ready to work with the Fashion-MNIST dataset in the sections that follow.

Summary

- 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 classification algorithms.
- We store the shape of any image with height h width w pixels as $h \times w$ or (h, w).
- Data iterators are a key component for efficient performance. Rely on well-implemented data iterators that exploit high-performance computing to avoid slowing down your training loop.

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.
3. Check out the framework's online API documentation. Which other datasets are available?

Discussions[57]

3.6 Implementation of Softmax Regression from Scratch

Just as we implemented linear regression from scratch, we believe that softmax regression is similarly fundamental and you ought to know the gory details of

how to implement it yourself. We will work with the Fashion-MNIST dataset, just introduced in Section 3.5, setting up a data iterator with batch size 256.

```
from IPython import display
from mxnet import autograd, gluon, np, npx
from d2l import mxnet as d2l

npx.set_np()
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

3.6.1 Initializing Model Parameters

As in our linear regression example, each example here will be represented by a fixed-length vector. Each example in the raw dataset is a 28×28 image. In this section, we will flatten each image, treating them as vectors of length 784. In the future, we will talk about more sophisticated strategies for exploiting the spatial structure in images, but for now we treat each pixel location as just another feature.

Recall that in softmax regression, we have as many outputs as there are classes. Because our dataset has 10 classes, our network will have an output dimension of 10. Consequently, our weights

[57] https://discuss.d2l.ai/t/48

will constitute a 784×10 matrix and the biases will constitute a 1×10 row vector. As with linear regression, we will initialize our weights `W` with Gaussian noise and our biases to take the initial value 0.

```
num_inputs = 784
num_outputs = 10

W = np.random.normal(0, 0.01, (num_inputs, num_outputs))
b = np.zeros(num_outputs)
W.attach_grad()
b.attach_grad()
```

3.6.2 Defining the Softmax Operation

Before implementing the softmax regression model, let us briefly review how the sum operator works along specific dimensions in a tensor, as discussed in Section 2.3.6 and Section 2.3.6. Given a matrix `X` we can sum over all elements (by default) or only over elements in the same axis, i.e., the same column (axis 0) or the same row (axis 1). Note that if `X` is a tensor with shape (2, 3) and we sum over the columns, the result will be a vector with shape (3,). When invoking the sum operator, we can specify to keep the number of axes in the original tensor, rather than collapsing out the dimension that we summed over. This will result in a two-dimensional tensor with shape (1, 3).

```
X = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdims=True), X.sum(1, keepdims=True)
```

```
(array([[5., 7., 9.]]),
 array([[ 6.],
        [15.]]))
```

We are now ready to implement the softmax operation. Recall that softmax consists of three steps: i) we exponentiate each term (using `exp`); ii) we sum over each row (we have one row per example in the batch) to get the normalization constant for each example; iii) we divide each row by its normalization constant, ensuring that the result sums to 1. Before looking at the code, let us recall how this looks expressed as an equation:

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}. \tag{3.6.1}$$

The denominator, or normalization constant, is also sometimes called the *partition function* (and its logarithm is called the log-partition function). The origins of that name are in statistical physics[58] where a related equation models the distribution over an ensemble of particles.

```
def softmax(X):
    X_exp = np.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition  # The broadcasting mechanism is applied here
```

As you can see, for any random input, we turn each element into a non-negative number. Moreover, each row sums up to 1, as is required for a probability.

[58] https://en.wikipedia.org/wiki/Partition_function_(statistical_mechanics)

```
X = np.random.normal(0, 1, (2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

```
(array([[0.22376052, 0.06659239, 0.06583703, 0.29964197, 0.3441681 ],
        [0.63209665, 0.03179282, 0.194987  , 0.09209415, 0.04902935]]),
 array([1.        , 0.99999994]))
```

Note that while this looks correct mathematically, we were a bit sloppy in our implementation because we failed to take precautions against numerical overflow or underflow due to large or very small elements of the matrix.

3.6.3 Defining the Model

Now that we have defined the softmax operation, we can implement the softmax regression model. The below code defines how the input is mapped to the output through the network. Note that we flatten each original image in the batch into a vector using the `reshape` function before passing the data through our model.

```
def net(X):
    return softmax(np.dot(X.reshape((-1, W.shape[0])), W) + b)
```

3.6.4 Defining the Loss Function

Next, we need to implement the cross-entropy loss function, as introduced in Section 3.4. This may be the most common loss function in all of deep learning because, at the moment, classification problems far outnumber regression problems.

Recall that cross-entropy takes the negative log-likelihood of the predicted probability assigned to the true label. Rather than iterating over the predictions with a Python for-loop (which tends to be inefficient), we can pick all elements by a single operator. Below, we create sample data `y_hat` with 2 examples of predicted probabilities over 3 classes and their corresponding labels `y`. With `y` we know that in the first example the first class is the correct prediction and in the second example the third class is the ground-truth. Using `y` as the indices of the probabilities in `y_hat`, we pick the probability of the first class in the first example and the probability of the third class in the second example.

```
y = np.array([0, 2])
y_hat = np.array([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
array([0.1, 0.5])
```

Now we can implement the cross-entropy loss function efficiently with just one line of code.

```
def cross_entropy(y_hat, y):
    return -np.log(y_hat[range(len(y_hat)), y])

cross_entropy(y_hat, y)
```

```
array([2.3025851, 0.6931472])
```

3.6.5 Classification 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. Gmail must categorize an email into "Primary", "Social", "Updates", or "Forums". 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 most about, and we will nearly always report it when training classifiers.

To compute accuracy we do the following. 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.

```
def accuracy(y_hat, y):  #@save
    """Compute the number of correct predictions."""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = y_hat.argmax(axis=1)
    cmp = y_hat.astype(y.dtype) == y
    return float(cmp.astype(y.dtype).sum())
```

We will continue to use the variables `y_hat` and `y` defined before as the predicted probability distributions and labels, respectively. We can see that the first example's prediction class is 2 (the largest element of the row is 0.6 with the index 2), which is inconsistent with the actual label, 0. The second example's prediction class is 2 (the largest element of the row is 0.5 with the index of 2), which is consistent with the actual label, 2. Therefore, the classification accuracy rate for these two examples is 0.5.

```
accuracy(y_hat, y) / len(y)
```

```
0.5
```

Similarly, we can evaluate the accuracy for any model `net` on a dataset that is accessed via the data iterator `data_iter`.

```
def evaluate_accuracy(net, data_iter):  #@save
    """Compute the accuracy for a model on a dataset."""
    metric = Accumulator(2)  # No. of correct predictions, no. of predictions
    for X, y in data_iter:
        metric.add(accuracy(net(X), y), y.size)
    return metric[0] / metric[1]
```

Here `Accumulator` is a utility class to accumulate sums over multiple variables. In the above `evaluate_accuracy` function, we create 2 variables in the `Accumulator` instance for storing both the number of correct predictions and the number of predictions, respectively. Both will be accumulated over time as we iterate over the dataset.

```
class Accumulator:  #@save
    """For accumulating sums over `n` variables."""
    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0.0] * len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]
```

Because we initialized the `net` model with random weights, the accuracy of this model should be close to random guessing, i.e., 0.1 for 10 classes.

```
evaluate_accuracy(net, test_iter)
```

```
0.0811
```

3.6.6 Training

The training loop for softmax regression should look strikingly familiar if you read through our implementation of linear regression in Section 3.2. Here we refactor the implementation to make it reusable. First, we define a function to train for one epoch. Note that `updater` is a general function to update the model parameters, which accepts the batch size as an argument. It can be either a wrapper of the `d2l.sgd` function or a framework's built-in optimization function.

```
def train_epoch_ch3(net, train_iter, loss, updater):  #@save
    """Train a model within one epoch (defined in Chapter 3)."""
    # Sum of training loss, sum of training accuracy, no. of examples
    metric = Accumulator(3)
    if isinstance(updater, gluon.Trainer):
        updater = updater.step
    for X, y in train_iter:
        # Compute gradients and update parameters
        with autograd.record():
            y_hat = net(X)
            l = loss(y_hat, y)
        l.backward()
        updater(X.shape[0])
        metric.add(float(l.sum()), accuracy(y_hat, y), y.size)
    # Return training loss and training accuracy
    return metric[0] / metric[2], metric[1] / metric[2]
```

Before showing the implementation of the training function, we define a utility class that plot data in animation. Again, it aims to simplify code in the rest of the book.

```
class Animator:  #@save
    """For plotting data in animation."""
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
                 figsize=(3.5, 2.5)):
        # Incrementally plot multiple lines
        if legend is None:
            legend = []
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes,]
        # Use a lambda function to capture arguments
        self.config_axes = lambda: d2l.set_axes(self.axes[
            0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        # Add multiple data points into the figure
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
        display.display(self.fig)
        display.clear_output(wait=True)
```

The following training function then trains a model `net` on a training dataset accessed via `train_iter` for multiple epochs, which is specified by `num_epochs`. At the end of each epoch, the model is evaluated on a testing dataset accessed via `test_iter`. We will leverage the `Animator` class to visualize the training progress.

```
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):  #@save
    """Train a model (defined in Chapter 3)."""
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                        legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics
    assert train_loss < 0.5, train_loss
```

(continues on next page)

(continued from previous page)

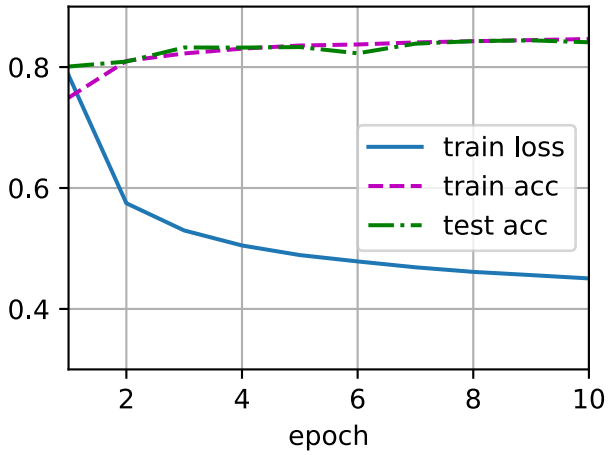
```
    assert train_acc <= 1 and train_acc > 0.7, train_acc
    assert test_acc <= 1 and test_acc > 0.7, test_acc
```

As an implementation from scratch, we use the minibatch stochastic gradient descent defined in Section 3.2 to optimize the loss function of the model with a learning rate 0.1.

```
lr = 0.1

def updater(batch_size):
    return d2l.sgd([W, b], lr, batch_size)
```

Now we train the model with 10 epochs. Note that both the number of epochs (`num_epochs`), and learning rate (`lr`) are adjustable hyperparameters. By changing their values, we may be able to increase the classification accuracy of the model.

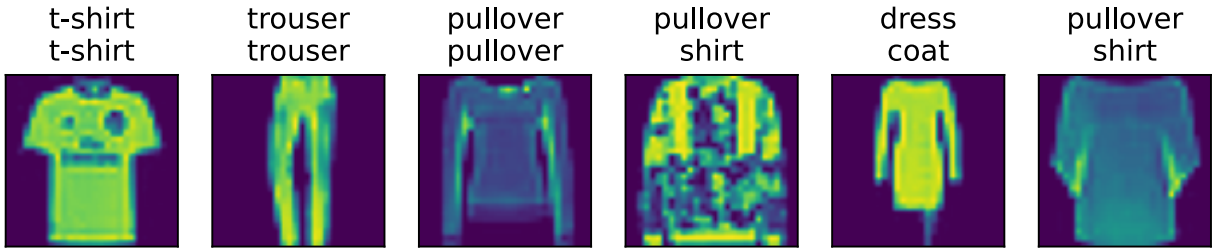
```
num_epochs = 10
train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)
```



3.6.7 Prediction

Now that training is complete, our model is ready to classify some images. Given a series of images, we will compare their actual labels (first line of text output) and the predictions from the model (second line of text output).

```
def predict_ch3(net, test_iter, n=6):  #@save
    """Predict labels (defined in Chapter 3)."""
    for X, y in test_iter:
        break
    trues = d2l.get_fashion_mnist_labels(y)
    preds = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1))
    titles = [true + '\n' + pred for true, pred in zip(trues, preds)]
    d2l.show_images(X[0:n].reshape((n, 28, 28)), 1, n, titles=titles[0:n])

predict_ch3(net, test_iter)
```



Summary

- With softmax regression, we can train models for multiclass classification.
- The training loop of softmax regression is very similar to that in linear regression: retrieve and read data, define models and loss functions, then train models using optimization algorithms. As you will soon find out, most common deep learning models have similar training procedures.

Exercises

1. In this section, we directly implemented the softmax function based on the mathematical definition of the softmax operation. What problems might this cause? Hint: try to calculate the size of $\exp(50)$.
2. The function `cross_entropy` in this section was implemented according to the definition of the cross-entropy loss function. What could be the problem with this implementation? Hint: consider the domain of the logarithm.
3. What solutions you can think of to fix the two problems above?
4. Is it always a good idea to return the most likely label? For example, would you do this for medical diagnosis?
5. 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?

Discussions[59]

3.7 Concise Implementation of Softmax Regression

Just as high-level APIs of deep learning frameworks made it much easier to implement linear regression in Section 3.3, we will find it similarly (or possibly more) convenient for implementing classification models. Let us stick with the Fashion-MNIST dataset and keep the batch size at 256 as in Section 3.6.

```
from mxnet import gluon, init, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

[59] https://discuss.d2l.ai/t/50

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

3.7.1 Initializing Model Parameters

As mentioned in Section 3.4, the output layer of softmax regression is a fully-connected layer. Therefore, to implement our model, we just need to add one fully-connected layer with 10 outputs to our `Sequential`. Again, here, the `Sequential` is not really necessary, but we might as well form the habit since it will be ubiquitous when implementing deep models. Again, we initialize the weights at random with zero mean and standard deviation 0.01.

```
net = nn.Sequential()
net.add(nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

3.7.2 Softmax Implementation Revisited

In the previous example of Section 3.6, we calculated our model's output and then ran this output through the cross-entropy loss. Mathematically, that is a perfectly reasonable thing to do. However, from a computational perspective, exponentiation can be a source of numerical stability issues.

Recall that the softmax function calculates $\hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$, where $\hat{y}_j$ is the j^{th} element of the predicted probability distribution $\hat{\mathbf{y}}$ and o_j is the j^{th} element of the logits $\mathbf{o}$. 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 (i.e., *overflow*). This would make the denominator (and/or numerator) `inf` (infinity) and we wind up encountering either 0, `inf`, or `nan` (not a number) for $\hat{y}_j$. In these situations we do not get a well-defined return value for cross-entropy.

One trick to get around this is to first subtract $\max(o_k)$ from all o_k before proceeding with the softmax calculation. You can see that this shifting of each o_k by constant factor does not change the return value of softmax:

$$\begin{aligned} \hat{y}_j &= \frac{\exp(o_j - \max(o_k))\exp(\max(o_k))}{\sum_k \exp(o_k - \max(o_k))\exp(\max(o_k))} \\ &= \frac{\exp(o_j - \max(o_k))}{\sum_k \exp(o_k - \max(o_k))}. \end{aligned} \tag{3.7.1}$$

After the subtraction and normalization step, it might be possible that some $o_j - \max(o_k)$ have large negative values and thus that the corresponding $\exp(o_j - \max(o_k))$ will take values close to zero. These might be rounded to zero due to finite precision (i.e., *underflow*), making $\hat{y}_j$ zero and giving us `-inf` for $\log(\hat{y}_j)$. A few steps down the road in backpropagation, we might find ourselves faced with a screenful of the dreaded `nan` 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 these two operators softmax and cross-entropy together, we can escape the numerical stability issues that might otherwise plague us during backpropagation. As shown in the equation below, we avoid calculating $\exp(o_j - \max(o_k))$ and can use instead $o_j - \max(o_k)$ directly due to the canceling

in $\log(\exp(\cdot))$:

$$\begin{aligned}\log\left(\hat{y}_j\right) &= \log\left(\frac{\exp(o_j - \max(o_k))}{\sum_k \exp(o_k - \max(o_k))}\right) \\ &= \log\left(\exp(o_j - \max(o_k))\right) - \log\left(\sum_k \exp(o_k - \max(o_k))\right) \\ &= o_j - \max(o_k) - \log\left(\sum_k \exp(o_k - \max(o_k))\right).\end{aligned} \tag{3.7.2}$$

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 will 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"[60].

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

3.7.3 Optimization Algorithm

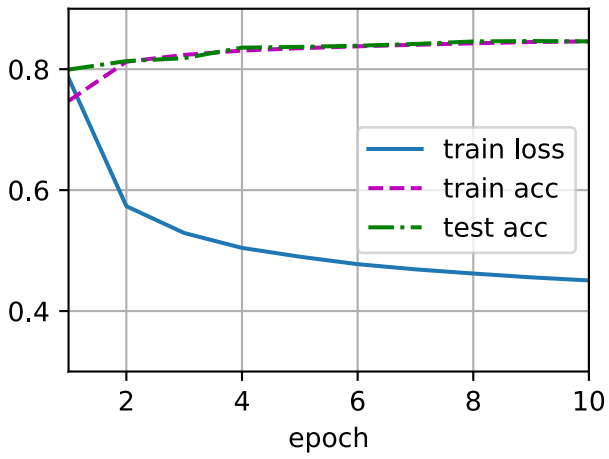
Here, we use minibatch stochastic gradient descent with a learning rate of 0.1 as the optimization algorithm. Note that this is the same as we applied in the linear regression example and it illustrates the general applicability of the optimizers.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

3.7.4 Training

Next we call the training function defined in Section 3.6 to train the model.

```
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



[60] https://en.wikipedia.org/wiki/LogSumExp

As before, this algorithm converges to a solution that achieves a decent accuracy, albeit this time with fewer lines of code than before.

Summary

- Using high-level APIs, we can implement softmax regression much more concisely.
- From a computational perspective, implementing softmax regression has intricacies. Note that in many cases, a deep learning framework takes additional precautions beyond these most well-known tricks to ensure numerical stability, saving us from even more pitfalls that we would encounter if we tried to code all of our models from scratch in practice.

Exercises

1. Try adjusting the hyperparameters, such as the batch size, number of epochs, and learning rate, to see what the results are.
2. Increase the number of epochs for training. Why might the test accuracy decrease after a while? How could we fix this?

Discussions[61]

[61] https://discuss.d2l.ai/t/52

4 | Multilayer Perceptrons

In this chapter, we will introduce your first truly *deep* network. The simplest deep networks are called multilayer perceptrons, and they consist of multiple layers of neurons each fully connected to those in the layer below (from which they receive input) and those above (which they, in turn, influence). When we train high-capacity models we run the risk of overfitting. Thus, we will need to provide your first rigorous introduction to the notions of overfitting, underfitting, and model selection. To help you combat these problems, we will introduce regularization techniques such as weight decay and dropout. We will also discuss issues relating to numerical stability and parameter initialization that are key to successfully training deep networks. Throughout, we aim to give you a firm grasp not just of the concepts but also of the practice of using deep networks. At the end of this chapter, we apply what we have introduced so far to a real case: house price prediction. We punt matters relating to the computational performance, scalability, and efficiency of our models to subsequent chapters.

4.1 Multilayer Perceptrons

In Chapter 3, we introduced softmax regression (Section 3.4), implementing the algorithm from scratch (Section 3.6) and using high-level APIs (Section 3.7), and training classifiers to recognize 10 categories of clothing from low-resolution images. Along the way, we learned how to wrangle data, coerce our outputs into a valid probability distribution, apply an appropriate loss function, and minimize it with respect to our model's parameters. Now that we have mastered these mechanics in the context of simple linear models, we can launch our exploration of deep neural networks, the comparatively rich class of models with which this book is primarily concerned.

4.1.1 Hidden Layers

We have described the affine transformation in Section 3.1.1, which is a linear transformation added by a bias. To begin, recall the model architecture corresponding to our softmax regression example, illustrated in Fig. 3.4.1. This model mapped our inputs directly to our outputs via a single affine transformation, followed by a softmax operation. If our labels truly were related to our input data by an affine transformation, then this approach would be sufficient. But linearity in affine transformations is a *strong* assumption.

Linear Models May Go Wrong

For example, linearity implies the *weaker* assumption of *monotonicity*: that any increase in our feature must either always cause an increase in our model's output (if the corresponding weight is positive), or always cause a decrease in our model's output (if the corresponding weight is negative). Sometimes that makes sense. For example, if we were trying to predict whether an individual will repay a loan, we might reasonably imagine that holding all else equal, an applicant with a higher income would always be more likely to repay than one with a lower income. While monotonic, this relationship likely is not linearly associated with the probability of repayment. An increase in income from 0 to 50 thousand likely corresponds to a bigger increase in likelihood of repayment than an increase from 1 million to 1.05 million. One way to handle this might be to preprocess our data such that linearity becomes more plausible, say, by using the logarithm of income as our feature.

Note that we can easily come up with examples that violate monotonicity. Say for example that we want to predict probability of death based on body temperature. For individuals with a body temperature above 37°C (98.6°F), higher temperatures indicate greater risk. However, for individuals with body temperatures below 37° C, higher temperatures indicate lower risk! In this case too, we might resolve the problem with some clever preprocessing. Namely, we might use the distance from 37°C as our feature.

But what about classifying images of cats and dogs? Should increasing the intensity of the pixel at location (13, 17) always increase (or always decrease) the likelihood that the image depicts a dog? Reliance on a linear model corresponds to the implicit assumption that the only requirement for differentiating cats vs. dogs is to assess the brightness of individual pixels. This approach is doomed to fail in a world where inverting an image preserves the category.

And yet despite the apparent absurdity of linearity here, as compared with our previous examples, it is less obvious that we could address the problem with a simple preprocessing fix. That is because the significance of any pixel depends in complex ways on its context (the values of the surrounding pixels). While there might exist a representation of our data that would take into account the relevant interactions among our features, on top of which a linear model would be suitable, we simply do not know how to calculate it by hand. With deep neural networks, we used observational data to jointly learn both a representation via hidden layers and a linear predictor that acts upon that representation.

Incorporating Hidden Layers

We can overcome these limitations of linear models and handle a more general class of functions by incorporating one or more hidden layers. The easiest way to do this is to stack many fully-connected layers on top of each other. Each layer feeds into the layer above it, until we generate outputs. We can think of the first $L-1$ layers as our representation and the final layer as our linear predictor. This architecture is commonly called a *multilayer perceptron*, often abbreviated as *MLP*. Below, we depict an MLP diagrammatically (Fig. 4.1.1).

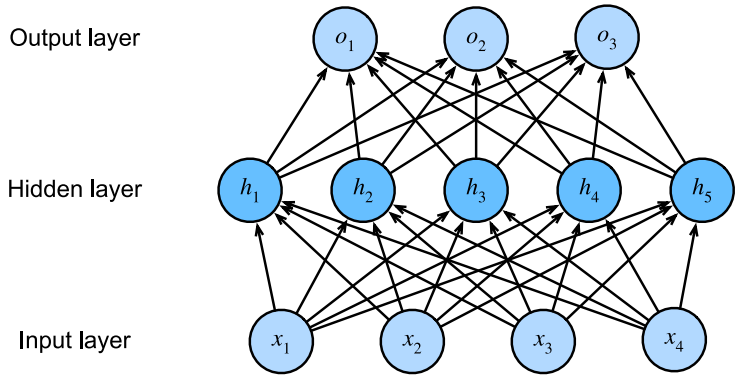


Fig. 4.1.1: An MLP with a hidden layer of 5 hidden units.

This MLP has 4 inputs, 3 outputs, and its hidden layer contains 5 hidden units. Since the input layer does not involve any calculations, producing outputs with this network requires implementing the computations for both the hidden and output layers; thus, the number of layers in this MLP is 2. Note that these layers are both fully connected. Every input influences every neuron in the hidden layer, and each of these in turn influences every neuron in the output layer. However, as suggested by Section 3.4.3, the parameterization cost of MLPs with fully-connected layers can be prohibitively high, which may motivate tradeoff between parameter saving and model effectiveness even without changing the input or output size (Zhang et al., 2021).

From Linear to Nonlinear

As before, by the matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, we denote a minibatch of n examples where each example has d inputs (features). For a one-hidden-layer MLP whose hidden layer has h hidden units, denote by $\mathbf{H} \in \mathbb{R}^{n \times h}$ the outputs of the hidden layer, which are *hidden representations*. In mathematics or code, $\mathbf{H}$ is also known as a *hidden-layer variable* or a *hidden variable*. Since the hidden and output layers are both fully connected, we have hidden-layer weights $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ and biases $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$ and output-layer weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ and biases $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$. Formally, we calculate the outputs $\mathbf{O} \in \mathbb{R}^{n \times q}$ of the one-hidden-layer MLP as follows:

$$\begin{aligned} \mathbf{H} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{4.1.1}$$

Note that after adding the hidden layer, our model now requires us to track and update additional sets of parameters. So what have we gained in exchange? You might be surprised to find out that—in the model defined above—*we gain nothing for our troubles*! The reason is plain. The hidden units above are given by an affine function of the inputs, and the outputs (pre-softmax) are just an affine function of the hidden units. An affine function of an affine function is itself an affine function. Moreover, our linear model was already capable of representing any affine function.

We can view the equivalence formally by proving that for any values of the weights, we can just collapse out the hidden layer, yielding an equivalent single-layer model with parameters $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ and $\mathbf{b} = \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$:

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}. \tag{4.1.2}$$

In order to realize the potential of multilayer architectures, we need one more key ingredient: a nonlinear *activation function* σ to be applied to each hidden unit following the affine transformation. The outputs of activation functions (e.g., $\sigma(\cdot)$) are called *activations*. In general, with

activation functions in place, it is no longer possible to collapse our MLP into a linear model:

$$\begin{aligned}\mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}),\\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.\end{aligned} \tag{4.1.3}$$

Since each row in $\mathbf{X}$ corresponds to an example in the minibatch, with some abuse of notation, we define the nonlinearity σ to apply to its inputs in a rowwise fashion, i.e., one example at a time. Note that we used the notation for softmax in the same way to denote a rowwise operation in Section 3.4.5. Often, as in this section, the activation functions that we apply to hidden layers are not merely rowwise, but elementwise. That means that after computing the linear portion of the layer, we can calculate each activation without looking at the values taken by the other hidden units. This is true for most activation functions.

To build more general MLPs, we can continue stacking such hidden layers, e.g., $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ and $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$, one atop another, yielding ever more expressive models.

Universal Approximators

MLPs can capture complex interactions among our inputs via their hidden neurons, which depend on the values of each of the inputs. We can easily design hidden nodes to perform arbitrary computation, for instance, basic logic operations on a pair of inputs. Moreover, for certain choices of the activation function, it is widely known that MLPs are universal approximators. Even with a single-hidden-layer network, given enough nodes (possibly absurdly many), and the right set of weights, we can model any function, though actually learning that function is the hard part. You might think of your neural network as being a bit like the C programming language. The language, like any other modern language, is capable of expressing any computable program. But actually coming up with a program that meets your specifications is the hard part.

Moreover, just because a single-hidden-layer network *can* learn any function does not mean that you should try to solve all of your problems with single-hidden-layer networks. In fact, we can approximate many functions much more compactly by using deeper (vs. wider) networks. We will touch upon more rigorous arguments in subsequent chapters.

4.1.2 Activation Functions

Activation functions decide whether a neuron should be activated or not by calculating the weighted sum and further adding bias with it. They are differentiable operators to transform input signals to outputs, while most of them add non-linearity. Because activation functions are fundamental to deep learning, let us briefly survey some common activation functions.

```
%matplotlib inline
from mxnet import autograd, np, npx
from d2l import mxnet as d2l

npx.set_np()
```

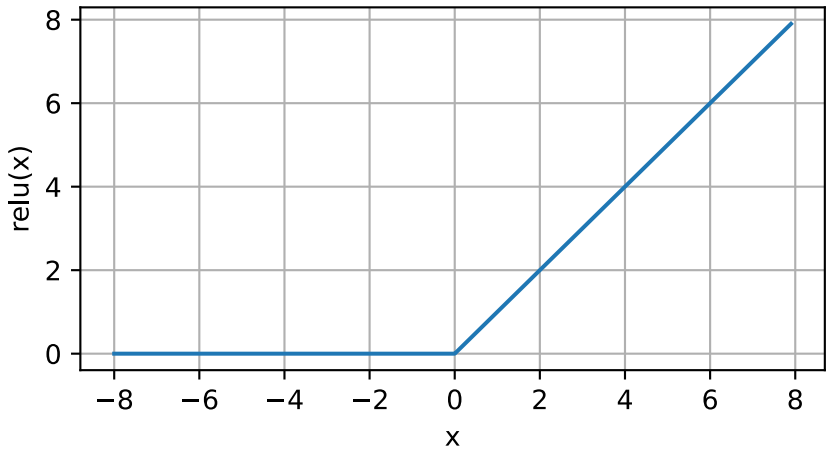
ReLU Function

The most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks, is the *rectified linear unit* (*ReLU*). ReLU provides a very simple nonlinear transformation. Given an element x, the function is defined as the maximum of that element and 0:

$$\text{ReLU}(x) = \max(x, 0). \tag{4.1.4}$$

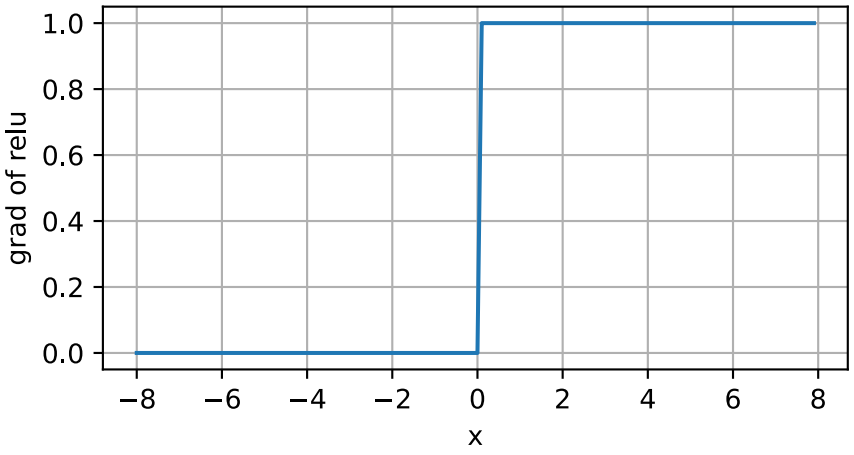
Informally, the ReLU function retains only positive elements and discards all negative elements by setting the corresponding activations to 0. To gain some intuition, we can plot the function. As you can see, the activation function is piecewise linear.

```
x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.relu(x)
d2l.plot(x, y, 'x', 'relu(x)', figsize=(5, 2.5))
```



When the input is negative, the derivative of the ReLU function is 0, and when the input is positive, the derivative of the ReLU function is 1. Note that the ReLU function is not differentiable when the input takes value precisely equal to 0. In these cases, we default to the left-hand-side derivative and say that the derivative is 0 when the input is 0. We can get away with this because the input may never actually be zero. There is an old adage that if subtle boundary conditions matter, we are probably doing (*real*) mathematics, not engineering. That conventional wisdom may apply here. We plot the derivative of the ReLU function plotted below.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



The reason for using ReLU is that its derivatives are particularly well behaved: either they vanish or they just let the argument through. This makes optimization better behaved and it mitigated the well-documented problem of vanishing gradients that plagued previous versions of neural networks (more on this later).

Note that there are many variants to the ReLU function, including the *parameterized ReLU* (*pReLU*) function (He et al., 2015). This variation adds a linear term to ReLU, so some information still gets through, even when the argument is negative:

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x). \tag{4.1.5}$$

Sigmoid Function

The *sigmoid function* transforms its inputs, for which values lie in the domain $\mathbb{R}$, to outputs that lie on the interval (0, 1). For that reason, the sigmoid is often called a *squashing function*: it squashes any input in the range (-inf, inf) to some value in the range (0, 1):

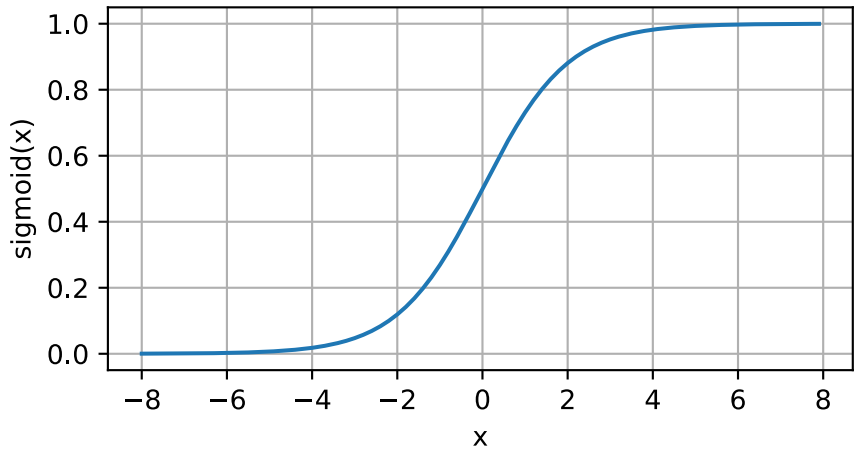
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{4.1.6}$$

In the earliest neural networks, scientists were interested in modeling biological neurons which either *fire* or *do not fire*. Thus the pioneers of this field, going all the way back to McCulloch and Pitts, the inventors of the artificial neuron, focused on thresholding units. A thresholding activation takes value 0 when its input is below some threshold and value 1 when the input exceeds the threshold.

When attention shifted to gradient based learning, the sigmoid function was a natural choice because it is a smooth, differentiable approximation to a thresholding unit. Sigmoids are still widely used as activation functions on the output units, when we want to interpret the outputs as probabilities for binary classification problems (you can think of the sigmoid as a special case of the softmax). However, the sigmoid has mostly been replaced by the simpler and more easily trainable ReLU for most use in hidden layers. In later chapters on recurrent neural networks, we will describe architectures that leverage sigmoid units to control the flow of information across time.

Below, we plot the sigmoid function. Note that when the input is close to 0, the sigmoid function approaches a linear transformation.

```
with autograd.record():
    y = npx.sigmoid(x)
d2l.plot(x, y, 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

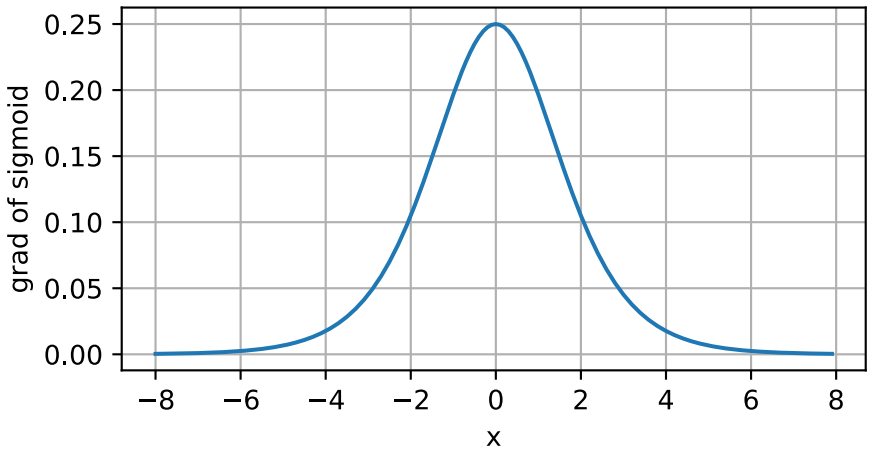


The derivative of the sigmoid function is given by the following equation:

$$\frac{d}{dx}\,\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x)\,(1-\text{sigmoid}(x))\,. \tag{4.1.7}$$

The derivative of the sigmoid function is plotted below. Note that when the input is 0, the derivative of the sigmoid function reaches a maximum of 0.25. As the input diverges from 0 in either direction, the derivative approaches 0.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



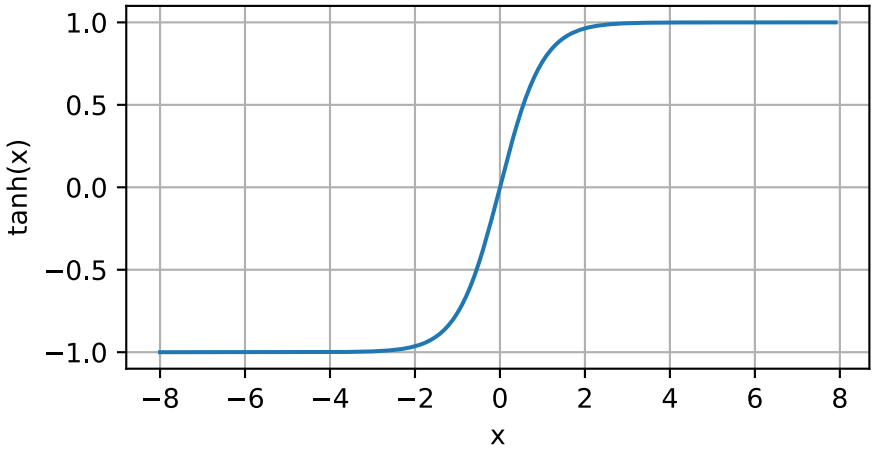
Tanh Function

Like the sigmoid function, the tanh (hyperbolic tangent) function also squashes its inputs, transforming them into elements on the interval between -1 and 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}. \tag{4.1.8}$$

We plot the tanh function below. Note that as the input nears 0, the tanh function approaches a linear transformation. Although the shape of the function is similar to that of the sigmoid function, the tanh function exhibits point symmetry about the origin of the coordinate system.

```
with autograd.record():
    y = np.tanh(x)
d2l.plot(x, y, 'x', 'tanh(x)', figsize=(5, 2.5))
```

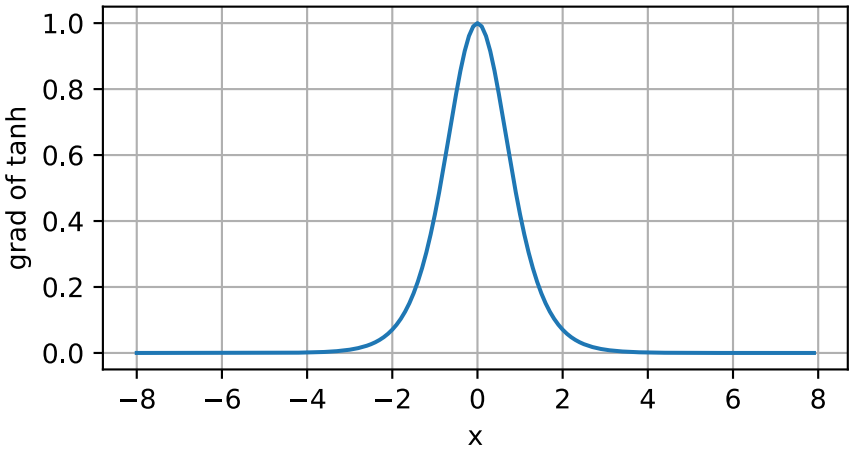


The derivative of the tanh function is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x). \tag{4.1.9}$$

The derivative of tanh function is plotted below. As the input nears 0, the derivative of the tanh function approaches a maximum of 1. And as we saw with the sigmoid function, as the input moves away from 0 in either direction, the derivative of the tanh function approaches 0.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



In summary, we now know how to incorporate nonlinearities to build expressive multilayer neural network architectures. As a side note, your knowledge already puts you in command of a similar toolkit to a practitioner circa 1990. In some ways, you have an advantage over anyone working in the 1990s, because you can leverage powerful open-source deep learning frameworks to build models rapidly, using only a few lines of code. Previously, training these networks required researchers to code up thousands of lines of C and Fortran.

Summary

- MLP adds one or multiple fully-connected hidden layers between the output and input layers and transforms the output of the hidden layer via an activation function.
- Commonly-used activation functions include the ReLU function, the sigmoid function, and the tanh function.

Exercises

1. Compute the derivative of the pReLU activation function.
2. Show that an MLP using only ReLU (or pReLU) constructs a continuous piecewise linear function.
3. Show that $\tanh(x) + 1 = 2 \,\text{sigmoid}(2x)$.
4. Assume that we have a nonlinearity that applies to one minibatch at a time. What kinds of problems do you expect this to cause?

Discussions[62]

[62] https://discuss.d2l.ai/t/90

4.2 Implementation of Multilayer Perceptrons from Scratch

Now that we have characterized multilayer perceptrons (MLPs) mathematically, let us try to implement one ourselves. To compare against our previous results achieved with softmax regression (Section 3.6), we will continue to work with the Fashion-MNIST image classification dataset (Section 3.5).

```
from mxnet import gluon, np, npx
from d2l import mxnet as d2l

npx.set_np()
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

4.2.1 Initializing Model Parameters

Recall that Fashion-MNIST contains 10 classes, and that each image consists of a $28 \times 28 = 784$ grid of grayscale pixel values. Again, we will disregard the spatial structure among the pixels for now, so we can think of this as simply a classification dataset with 784 input features and 10 classes. To begin, we will implement an MLP with one hidden layer and 256 hidden units. Note that we can regard both of these quantities as hyperparameters. Typically, we choose layer widths in powers of 2, which tend to be computationally efficient because of how memory is allocated and addressed in hardware.

Again, we will represent our parameters with several tensors. Note that *for every layer*, we must keep track of one weight matrix and one bias vector. As always, we allocate memory for the gradients of the loss with respect to these parameters.

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = np.random.normal(scale=0.01, size=(num_inputs, num_hiddens))
b1 = np.zeros(num_hiddens)
W2 = np.random.normal(scale=0.01, size=(num_hiddens, num_outputs))
b2 = np.zeros(num_outputs)
params = [W1, b1, W2, b2]

for param in params:
    param.attach_grad()
```

4.2.2 Activation Function

To make sure we know how everything works, we will implement the ReLU activation ourselves using the maximum function rather than invoking the built-in `relu` function directly.

```
def relu(X):
    return np.maximum(X, 0)
```

4.2.3 Model

Because we are disregarding spatial structure, we `reshape` each two-dimensional image into a flat vector of length `num_inputs`. Finally, we implement our model with just a few lines of code.

```
def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(np.dot(X, W1) + b1)
    return np.dot(H, W2) + b2
```

4.2.4 Loss Function

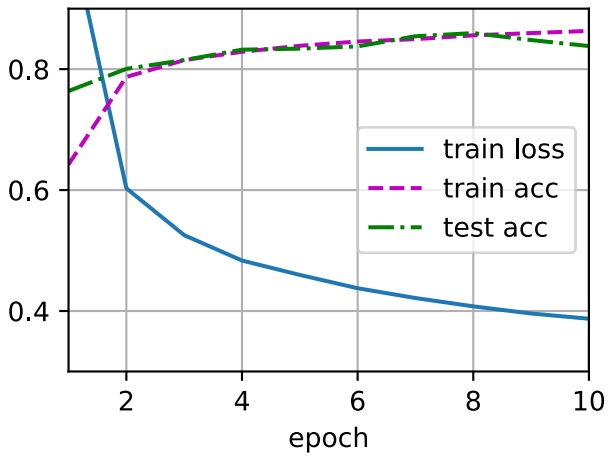
To ensure numerical stability, and because we already implemented the softmax function from scratch (Section 3.6), we leverage the integrated function from high-level APIs for calculating the softmax and cross-entropy loss. Recall our earlier discussion of these intricacies in Section 3.7.2. We encourage the interested reader to examine the source code for the loss function to deepen their knowledge of implementation details.

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

4.2.5 Training

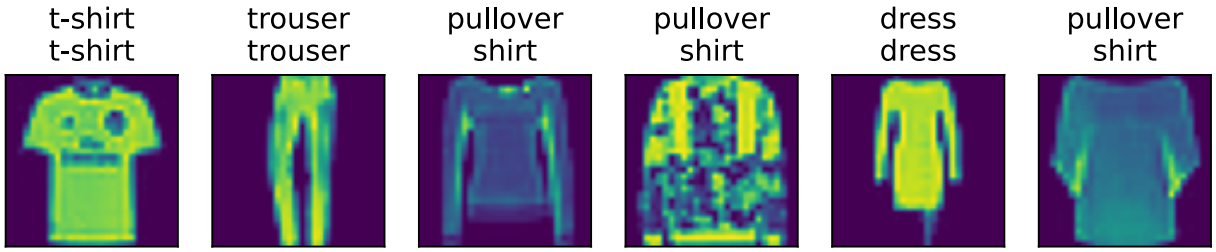
Fortunately, the training loop for MLPs is exactly the same as for softmax regression. Leveraging the `d2l` package again, we call the `train_ch3` function (see Section 3.6), setting the number of epochs to 10 and the learning rate to 0.1.

```
num_epochs, lr = 10, 0.1
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
              lambda batch_size: d2l.sgd(params, lr, batch_size))
```



To evaluate the learned model, we apply it on some test data.

```
d2l.predict_ch3(net, test_iter)
```



Summary

- We saw that implementing a simple MLP is easy, even when done manually.
- However, with a large number of layers, implementing MLPs from scratch can still get messy (e.g., naming and keeping track of our model's parameters).

Exercises

1. Change the value of the hyperparameter `num_hiddens` and see how this hyperparameter influences your results. Determine the best value of this hyperparameter, keeping all others constant.
2. Try adding an additional hidden layer to see how it affects the results.
3. How does changing the learning rate alter your results? Fixing the model architecture and other hyperparameters (including number of epochs), what learning rate gives you the best results?
4. What is the best result you can get by optimizing over all the hyperparameters (learning rate, number of epochs, number of hidden layers, number of hidden units per layer) jointly?
5. Describe why it is much more challenging to deal with multiple hyperparameters.
6. What is the smartest strategy you can think of for structuring a search over multiple hyperparameters?

Discussions[63]

4.3 Concise Implementation of Multilayer Perceptrons

As you might expect, by relying on the high-level APIs, we can implement MLPs even more concisely.

```
from mxnet import gluon, init, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

[63] https://discuss.d2l.ai/t/92

4.3.1 Model

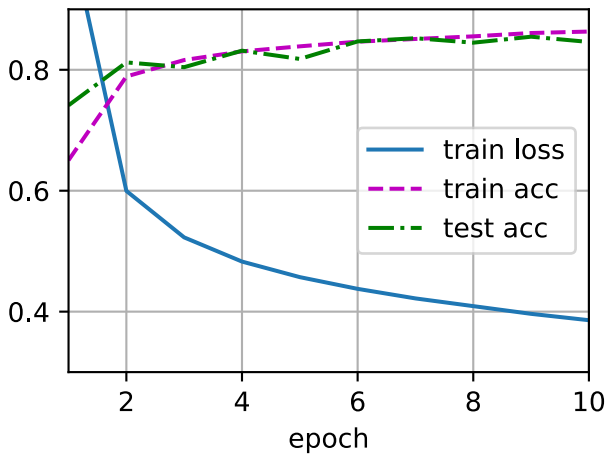
As compared with our concise implementation of softmax regression implementation (Section 3.7), the only difference is that we add *two* fully-connected layers (previously, we added *one*). The first is our hidden layer, which contains 256 hidden units and applies the ReLU activation function. The second is our output layer.

```
net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'), nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

The training loop is exactly the same as when we implemented softmax regression. This modularity enables us to separate matters concerning the model architecture from orthogonal considerations.

```
batch_size, lr, num_epochs = 256, 0.1, 10
loss = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
```

```
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



Summary

- Using high-level APIs, we can implement MLPs much more concisely.
- For the same classification problem, the implementation of an MLP is the same as that of softmax regression except for additional hidden layers with activation functions.

Exercises

1. Try adding different numbers of hidden layers (you may also modify the learning rate). What setting works best?
2. Try out different activation functions. Which one works best?
3. Try different schemes for initializing the weights. What method works best?

Discussions[64]

4.4 Model Selection, Underfitting, and Overfitting

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? For example, imagine that we wanted to hunt for patterns among genetic markers linking patients to their dementia status, where the labels are drawn from the set {dementia, mild cognitive impairment, healthy}. Because each person's genes identify them uniquely (ignoring identical siblings), it is possible to memorize the entire dataset.

We do not want our model to say *"That's Bob! I remember him! He has dementia!"* The reason why is simple. When we deploy the model in the future, we will encounter patients that the model has never seen before. Our predictions will only be useful if our model has truly discovered a *general* pattern.

To recapitulate more formally, our goal is to discover patterns that capture regularities in the underlying population from which our training set was drawn. If we are successful in this endeavor, then we could successfully assess risk even for individuals that we have never encountered before. This problem—how to discover patterns that *generalize*—is the fundamental problem of machine learning.

The danger is that when we train models, we access just a small sample of data. The largest public image datasets contain roughly one million images. More often, we must learn from only thousands or tens of thousands of data examples. In a large hospital system, we might access hundreds of thousands of medical records. When working with finite samples, we run the risk that we might discover apparent associations that turn out not to hold up when we collect more data.

The phenomenon of fitting our training data more closely than we fit the underlying distribution is called *overfitting*, and the techniques used to combat overfitting are called *regularization*. In the previous sections, you might have observed this effect while experimenting with the Fashion-MNIST dataset. If you altered the model structure or the hyperparameters during the experiment, you might have noticed that with enough neurons, layers, and training epochs, the model can eventually reach perfect accuracy on the training set, even as the accuracy on test data deteriorates.

[64] https://discuss.d2l.ai/t/94

4.4.1 Training Error and Generalization Error

In order to discuss this phenomenon more formally, we need to differentiate between training error and generalization error. The *training error* is the error of our model as calculated on the training dataset, while *generalization error* is the expectation of our model's error were we to apply it to an infinite stream of additional data examples drawn from the same underlying data distribution as our original sample.

Problematically, we can never calculate the generalization error exactly. That is because the stream of infinite data is an imaginary object. In practice, we must *estimate* the generalization error by applying our model to an independent test set constituted of a random selection of data examples that were withheld from our training set.

The following three thought experiments will help illustrate this situation better. Consider a college student trying to prepare for his final exam. A diligent student will strive to practice well and test his abilities using exams from previous years. Nonetheless, doing well on past exams is no guarantee that he will excel when it matters. For instance, the student might try to prepare by rote learning the answers to the exam questions. This requires the student to memorize many things. She might even remember the answers for past exams perfectly. Another student might prepare by trying to understand the reasons for giving certain answers. In most cases, the latter student will do much better.

Likewise, consider a model that simply uses a lookup table to answer questions. If the set of allowable inputs is discrete and reasonably small, then perhaps after viewing *many* training examples, this approach would perform well. Still this model has no ability to do better than random guessing when faced with examples that it has never seen before. In reality the input spaces are far too large to memorize the answers corresponding to every conceivable input. For example, consider the black and white 28×28 images. If each pixel can take one among 256 grayscale values, then there are 256^{784} possible images. That means that there are far more low-resolution grayscale thumbnail-sized images than there are atoms in the universe. Even if we could encounter such data, we could never afford to store the lookup table.

Last, consider the problem of trying to classify the outcomes of coin tosses (class 0: heads, class 1: tails) based on some contextual features that might be available. Suppose that the coin is fair. No matter what algorithm we come up with, the generalization error will always be $\frac{1}{2}$. However, for most algorithms, we should expect our training error to be considerably lower, depending on the luck of the draw, even if we did not have any features! Consider the dataset {0, 1, 1, 1, 0, 1}. Our feature-less algorithm would have to fall back on always predicting the *majority class*, which appears from our limited sample to be *1*. In this case, the model that always predicts class 1 will incur an error of $\frac{1}{3}$, considerably better than our generalization error. As we increase the amount of data, the probability that the fraction of heads will deviate significantly from $\frac{1}{2}$ diminishes, and our training error would come to match the generalization error.

Since generalization is the fundamental problem in machine learning, you might not be surprised to learn that many mathematicians and theorists have dedicated their lives to developing formal theories to describe this phenomenon. In their eponymous theorem[65], Glivenko and Cantelli derived the rate at which the training error converges to the generalization error. In a series of seminal papers, Vapnik and Chervonenkis[66] extended this theory to more general classes of functions. This work laid the foundations of statistical learning theory.

In the standard supervised learning setting, which we have addressed up until now and will stick with throughout most of this book, we assume that both the training data and the test data are drawn *independently* from *identical* distributions. This is commonly called the *i.i.d. assumption*, which means that the process that samples our data has no memory. In other words, the second example drawn and the third drawn are no more correlated than the second and the two-millionth sample drawn.

Being a good machine learning scientist requires thinking critically, and already you should be poking holes in this assumption, coming up with common cases where the assumption fails. What if we train a mortality risk predictor on data collected from patients at UCSF Medical Center, and apply it on patients at Massachusetts General Hospital? These distributions are simply not identical. Moreover, draws might be correlated in time. What if we are classifying the topics of Tweets? The news cycle would create temporal dependencies in the topics being discussed, violating any assumptions of independence.

Sometimes we can get away with minor violations of the i.i.d. assumption and our models will continue to work remarkably well. After all, nearly every real-world application involves at least some minor violation of the i.i.d. assumption, and yet we have many useful tools for various applications such as face recognition, speech recognition, and language translation.

Other violations are sure to cause trouble. Imagine, for example, if we try to train a face recognition system by training it exclusively on university students and then want to deploy it as a tool for monitoring geriatrics in a nursing home population. This is unlikely to work well since college students tend to look considerably different from the elderly.

In subsequent chapters, we will discuss problems arising from violations of the i.i.d. assumption. For now, even taking the i.i.d. assumption for granted, understanding generalization is a formidable problem. Moreover, elucidating the precise theoretical foundations that might explain why deep neural networks generalize as well as they do continues to vex the greatest minds in learning theory.

When we train our models, we attempt to search for a function that fits the training data as well as possible. If the function is so flexible that it can catch on to spurious patterns just as easily as to true associations, then it might perform *too well* without producing a model that generalizes well to unseen data. This is precisely what we want to avoid or at least control. Many of the techniques in deep learning are heuristics and tricks aimed at guarding against overfitting.

[65] https://en.wikipedia.org/wiki/Glivenko%E2%80%93Cantelli_theorem
[66] https://en.wikipedia.org/wiki/Vapnik%E2%80%93Chervonenkis_theory

Model Complexity

When we have simple models and abundant data, we expect the generalization error to resemble the training error. When we work with more complex models and fewer examples, we expect the training error to go down but the generalization gap to grow. What precisely constitutes model complexity is a complex matter. Many factors govern whether a model will generalize well. For example a model with more parameters might be considered more complex. A model whose parameters can take a wider range of values might be more complex. Often with neural networks, we think of a model that takes more training iterations as more complex, and one subject to *early stopping* (fewer training iterations) as less complex.

It can be difficult to compare the complexity among members of substantially different model classes (say, decision trees vs. neural networks). For now, a simple rule of thumb is quite useful: a model that can readily explain arbitrary facts is what statisticians view as complex, whereas one that has only a limited expressive power but still manages to explain the data well is probably closer to the truth. In philosophy, this is closely related to Popper's criterion of falsifiability of a scientific theory: a theory is good if it fits data and if there are specific tests that can be used to disprove it. This is important since all statistical estimation is *post hoc*, i.e., we estimate after we observe the facts, hence vulnerable to the associated fallacy. For now, we will put the philosophy aside and stick to more tangible issues.

In this section, to give you some intuition, we will focus on a few factors that tend to influence the generalizability of a model class:

1. The number of tunable parameters. When the number of tunable parameters, sometimes called the *degrees of freedom*, is large, models tend to be more susceptible to overfitting.
2. The values taken by the parameters. When weights can take a wider range of values, models can be more susceptible to overfitting.
3. The number of training examples. It is trivially easy to overfit a dataset containing only one or two examples even if your model is simple. But overfitting a dataset with millions of examples requires an extremely flexible model.

4.4.2 Model Selection

In machine learning, we usually select our final model after evaluating several candidate models. This process is called *model selection*. Sometimes the models subject to comparison are fundamentally different in nature (say, decision trees vs. linear models). At other times, we are comparing members of the same class of models that have been trained with different hyperparameter settings.

With MLPs, for example, we may wish to compare models with different numbers of hidden layers, different numbers of hidden units, and various choices of the activation functions applied to each hidden layer. In order to determine the best among our candidate models, we will typically employ a validation dataset.

Validation Dataset

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?

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 to 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.

The common practice to address this problem is to split our data three ways, incorporating a *validation dataset* (or *validation set*) in addition to the training and test datasets. The result is a murky practice 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.

K-Fold 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.

4.4.3 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* between our training and validation 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. With deep learning especially, it is well known that the best predictive models often perform far better on training data than on holdout data. Ultimately, we usually care more about the validation error than about the gap between the training and validation errors.

Whether we overfit or underfit can depend both on the complexity of our model and the size of the available training datasets, two topics that we discuss below.

Model Complexity

To illustrate some classical intuition about overfitting and model complexity, we give an example using polynomials. 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

$$\hat{y} = \sum_{i=0}^{d} x^i w_i \tag{4.4.1}$$

to estimate the labels 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 the data examples each have a distinct value of x, a polynomial function with degree equal to the number of data examples can fit the training set perfectly. We visualize the relationship between polynomial degree and underfitting vs. overfitting in Fig. 4.4.1.

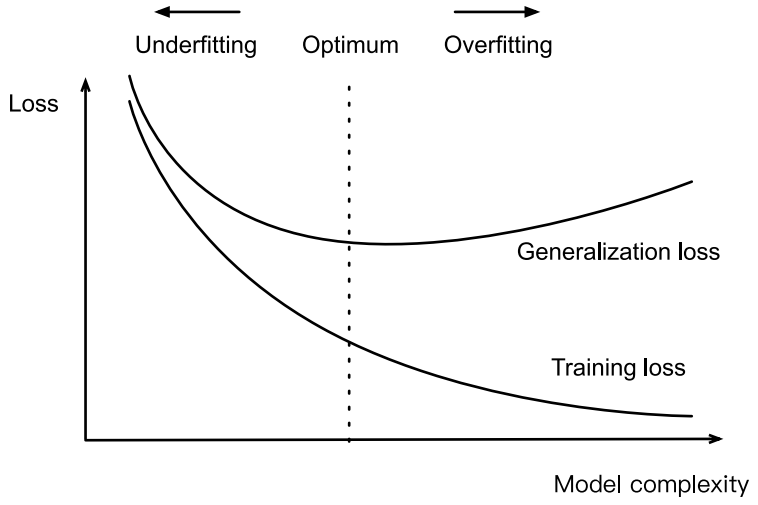


Fig. 4.4.1: Influence of model complexity on underfitting and overfitting

Dataset Size

The other big consideration to bear in mind is the 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 hurt. For a fixed task and data distribution, there is typically a relationship between model complexity and dataset size. Given more data, we might profitably 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 to the current abundance of massive datasets due to Internet companies, cheap storage, connected devices, and the broad digitization of the economy.

4.4.4 Polynomial Regression

We can now explore these concepts interactively by fitting polynomials to data.

```
import math
from mxnet import gluon, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

Generating the Dataset

First we need data. Given x, we will use the following cubic polynomial to generate the labels on training and test data:

$$y = 5 + 1.2x - 3.4\frac{x^2}{2!} + 5.6\frac{x^3}{3!} + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1^2). \tag{4.4.2}$$

The noise term ϵ obeys a normal distribution with a mean of 0 and a standard deviation of 0.1. For optimization, we typically want to avoid very large values of gradients or losses. This is why the *features* are rescaled from x^i to $\frac{x^i}{i!}$. It allows us to avoid very large values for large exponents i. We will synthesize 100 samples each for the training set and test set.

```
max_degree = 20  # Maximum degree of the polynomial
n_train, n_test = 100, 100  # Training and test dataset sizes
true_w = np.zeros(max_degree)  # Allocate lots of empty space
true_w[0:4] = np.array([5, 1.2, -3.4, 5.6])

features = np.random.normal(size=(n_train + n_test, 1))
np.random.shuffle(features)
poly_features = np.power(features, np.arange(max_degree).reshape(1, -1))
for i in range(max_degree):
    poly_features[:, i] /= math.gamma(i + 1)  # `gamma(n)` = (n-1)!
# Shape of `labels`: (`n_train` + `n_test`,)
labels = np.dot(poly_features, true_w)
labels += np.random.normal(scale=0.1, size=labels.shape)
```

Again, monomials stored in `poly_features` are rescaled by the gamma function, where $\Gamma(n) = (n-1)!$. Take a look at the first 2 samples from the generated dataset. The value 1 is technically a feature, namely the constant feature corresponding to the bias.

```
features[:2], poly_features[:2, :], labels[:2]
```

```
(array([[-0.03716067],
        [-1.1468065 ]]),
 array([[ 1.0000000e+00, -3.7160669e-02,  6.9045764e-04, -8.5526226e-06,
          7.9455290e-08, -5.9052235e-10,  3.6573678e-12, -1.9415747e-14,
          9.0187767e-17, -3.7238198e-19,  1.3837962e-21, -4.6747992e-24,
          1.4476556e-26, -4.1381425e-29,  1.0984010e-31, -2.7211542e-34,
          6.3199942e-37, -1.3815009e-39,  2.8516424e-42, -5.6051939e-45],
        [ 1.0000000e+00, -1.1468065e+00,  6.5758252e-01, -2.5137332e-01,
          7.2069131e-02, -1.6529869e-02,  3.1594271e-03, -5.1760738e-04,
          7.4199430e-05, -9.4547095e-06,  1.0842722e-06, -1.1304095e-07,
          1.0803007e-08, -9.5299690e-10,  7.8064499e-11, -5.9683248e-12,
          4.2778208e-13, -2.8857840e-14,  1.8385756e-15, -1.1097316e-16]]),
 array([ 5.1432443 , -0.06415121]))
```

Training and Testing the Model

Let us first implement a function to evaluate the loss on a given dataset.

```
def evaluate_loss(net, data_iter, loss):  #@save
    """Evaluate the loss of a model on the given dataset."""
    metric = d2l.Accumulator(2)  # Sum of losses, no. of examples
    for X, y in data_iter:
        l = loss(net(X), y)
        metric.add(l.sum(), l.size)
    return metric[0] / metric[1]
```

Now define the training function.

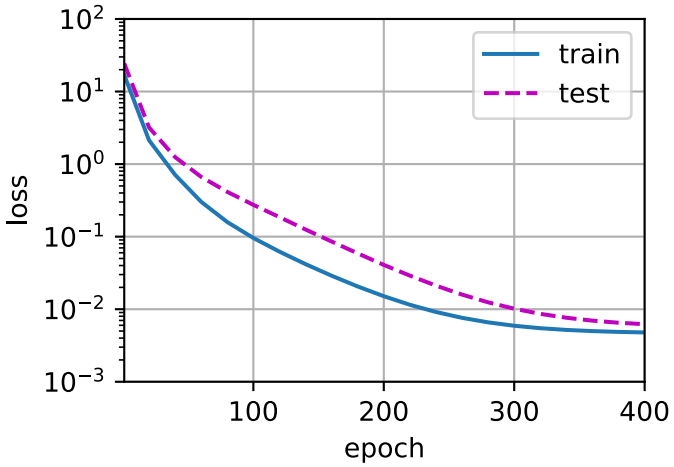
```
def train(train_features, test_features, train_labels, test_labels,
          num_epochs=400):
    loss = gluon.loss.L2Loss()
    net = nn.Sequential()
    # Switch off the bias since we already catered for it in the polynomial
    # features
    net.add(nn.Dense(1, use_bias=False))
    net.initialize()
    batch_size = min(10, train_labels.shape[0])
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    test_iter = d2l.load_array((test_features, test_labels), batch_size,
                               is_train=False)
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': 0.01})
    animator = d2l.Animator(xlabel='epoch', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], ylim=[1e-3, 1e2],
                            legend=['train', 'test'])
    for epoch in range(num_epochs):
        d2l.train_epoch_ch3(net, train_iter, loss, trainer)
        if epoch == 0 or (epoch + 1) % 20 == 0:
            animator.add(epoch + 1, (evaluate_loss(
                net, train_iter, loss), evaluate_loss(net, test_iter, loss)))
    print('weight:', net[0].weight.data().asnumpy())
```

Third-Order Polynomial Function Fitting (Normal)

We will begin by first using a third-order polynomial function, which is the same order as that of the data generation function. The results show that this model's training and test losses can be both effectively reduced. The learned model parameters are also close to the true values $w = [5, 1.2, -3.4, 5.6]$.

```
# Pick the first four dimensions, i.e., 1, x, x^2/2!, x^3/3! from the
# polynomial features
train(poly_features[:n_train, :4], poly_features[n_train:, :4],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 5.019045   1.2219346 -3.4237804  5.5718646]]
```

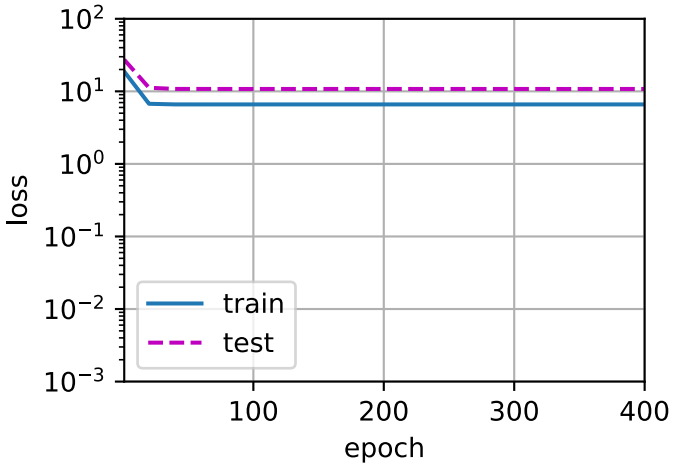


Linear Function Fitting (Underfitting)

Let us take another look at linear function fitting. After the decline in early epochs, it becomes difficult to further decrease this model's training loss. After the last epoch iteration has been completed, the training loss is still high. When used to fit nonlinear patterns (like the third-order polynomial function here) linear models are liable to underfit.

```
# Pick the first two dimensions, i.e., 1, x, from the polynomial features
train(poly_features[:n_train, :2], poly_features[n_train:, :2],
      labels[:n_train], labels[n_train:])
```

```
weight: [[2.6992648 4.2271123]]
```

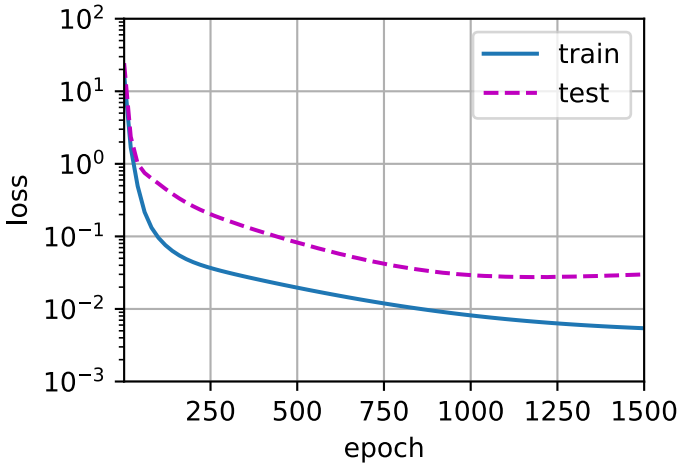


Higher-Order Polynomial Function Fitting (Overfitting)

Now let us try to train the model using a polynomial of too high degree. Here, there are insufficient data to learn that the higher-degree coefficients should have values close to zero. As a result, our overly-complex model is so susceptible that it is being influenced by noise in the training data. Though the training loss can be effectively reduced, the test loss is still much higher. It shows that the complex model overfits the data.

```
# Pick all the dimensions from the polynomial features
train(poly_features[:n_train, :], poly_features[n_train:, :],
      labels[:n_train], labels[n_train:], num_epochs=1500)
```

```
weight: [[ 4.992137    1.3060913  -3.3531141   5.1165624  -0.1113829   1.3031913
   0.12666036  0.16659527  0.05126056 -0.02273984  0.00805709 -0.05167707
  -0.02426345 -0.01502198 -0.0494136   0.06389865 -0.04761846 -0.04380166
  -0.05188227  0.05655775]]
```



In the subsequent sections, we will continue to discuss overfitting problems and methods for dealing with them, such as weight decay and dropout.

Summary

- Since the generalization error cannot be estimated based on the training error, simply minimizing the training error will not necessarily mean a reduction in the generalization error. Machine learning models need to be careful to safeguard against overfitting so as to minimize the generalization error.
- A validation set can be used for model selection, provided that it is not used too liberally.
- Underfitting means that a model is not able to reduce the training error. When training error is much lower than validation error, there is overfitting.
- We should choose an appropriately complex model and avoid using insufficient training samples.

Exercises

1. Can you solve the polynomial regression problem exactly? Hint: use linear algebra.
2. Consider model selection for polynomials:
    1. Plot the training loss vs. model complexity (degree of the polynomial). What do you observe? What degree of polynomial do you need to reduce the training loss to 0?
    2. Plot the test loss in this case.
    3. Generate the same plot as a function of the amount of data.
3. What happens if you drop the normalization ($1/i!$) of the polynomial features x^i? Can you fix this in some other way?
4. Can you ever expect to see zero generalization error?

Discussions[67]

4.5 Weight Decay

Now that we have characterized the problem of overfitting, we can introduce some standard techniques for regularizing models. Recall that we can always mitigate overfitting by going out and collecting more training data. 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 on regularization techniques.

Recall that in our polynomial regression example (Section 4.4) we could limit our model's capacity simply by tweaking the degree of the fitted polynomial. Indeed, limiting the number of features is a popular technique to mitigate overfitting. However, simply tossing aside features can be too blunt an instrument for the job. Sticking with the polynomial regression example, consider what might happen with high-dimensional inputs. 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.

[67] https://discuss.d2l.ai/t/96

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 (i.e., k multichoose 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.

4.5.1 Norms and Weight Decay

We have described both the L_2 norm and the L_1 norm, which are special cases of the more general L_p norm in Section 2.3.10. *Weight decay* (commonly called L_2 regularization), 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 its distance 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 answering this issue.

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., $\|\mathbf{w}\|^2$. 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 $\|\mathbf{w}\|^2$ vs. minimizing the training error. That is exactly what we want. To illustrate things in code, let us revive our previous example from Section 3.1 for linear regression. There, our loss was given by

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{4.5.1}$$

Recall that $\mathbf{x}^{(i)}$ are the features, $y^{(i)}$ are labels for all data examples 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 tradeoff via the *regularization constant* λ, a nonnegative hyperparameter that we fit using validation data:

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2, \tag{4.5.2}$$

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 model in statistics, which is 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 is called *feature selection*, which may be desirable for other reasons.

Using the same notation in (3.1.10), the minibatch stochastic gradient descent updates for L_2-regularized regression follow:

$$\mathbf{w} \leftarrow (1 - \eta\lambda)\,\mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\mathbf{x}^{(i)}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b - y^{(i)}\right). \tag{4.5.3}$$

As before, we update **w** based on the amount by which our estimate differs from the observation. However, we also shrink the size of **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 continuous mechanism for adjusting the complexity of a function. Smaller values of λ correspond to less constrained **w**, whereas larger values of λ constrain **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 of a network's output layer.

4.5.2 High-Dimensional Linear Regression

We can illustrate the benefits of weight decay through a simple synthetic example.

```
%matplotlib inline
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

First, we generate some data as before

$$y = 0.05 + \sum_{i=1}^{d} 0.01x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01^2). \tag{4.5.4}$$

We choose our label to be a linear function of our inputs, corrupted by Gaussian noise with zero mean and standard deviation 0.01. To make the effects of overfitting pronounced, we can increase the dimensionality of our problem to $d = 200$ and work with a small training set containing only 20 examples.

```
n_train, n_test, num_inputs, batch_size = 20, 100, 200, 5
true_w, true_b = np.ones((num_inputs, 1)) * 0.01, 0.05
train_data = d2l.synthetic_data(true_w, true_b, n_train)
train_iter = d2l.load_array(train_data, batch_size)
test_data = d2l.synthetic_data(true_w, true_b, n_test)
test_iter = d2l.load_array(test_data, batch_size, is_train=False)
```

4.5.3 Implementation from Scratch

In the following, we will implement weight decay from scratch, simply by adding the squared L_2 penalty to the original target function.

Initializing Model Parameters

First, we will define a function to randomly initialize our model parameters.

```
def init_params():
    w = np.random.normal(scale=1, size=(num_inputs, 1))
    b = np.zeros(1)
    w.attach_grad()
    b.attach_grad()
    return [w, b]
```

Defining L_2 Norm Penalty

Perhaps the most convenient way to implement this penalty is to square all terms in place and sum them up.

```
def l2_penalty(w):
    return (w**2).sum() / 2
```

Defining the Training Loop

The following code fits a model on the training set and evaluates it on the test set. The linear network and the squared loss have not changed since Chapter 3, so we will just import them via `d2l.linreg` and `d2l.squared_loss`. The only change here is that our loss now includes the penalty term.

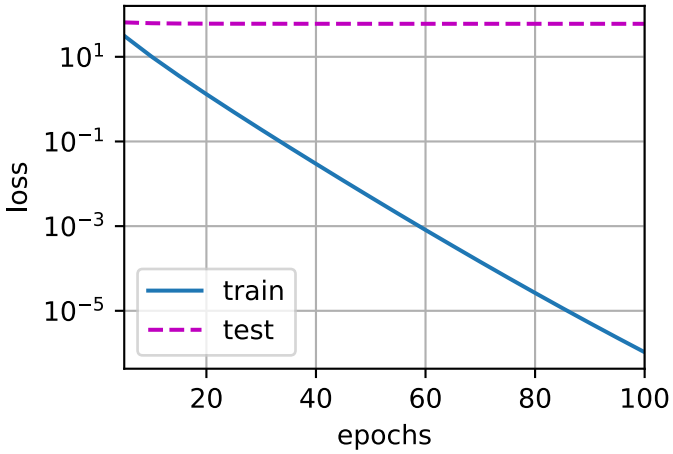
```
def train(lambd):
    w, b = init_params()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    num_epochs, lr = 100, 0.003
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                # The L2 norm penalty term has been added, and broadcasting
                # makes `l2_penalty(w)` a vector whose length is `batch_size`
                l = loss(net(X), y) + lambd * l2_penalty(w)
            l.backward()
            d2l.sgd([w, b], lr, batch_size)
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('L2 norm of w:', np.linalg.norm(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 test error—a textbook case of overfitting.

```
train(lambd=0)
```

```
L2 norm of w: 13.259389
```

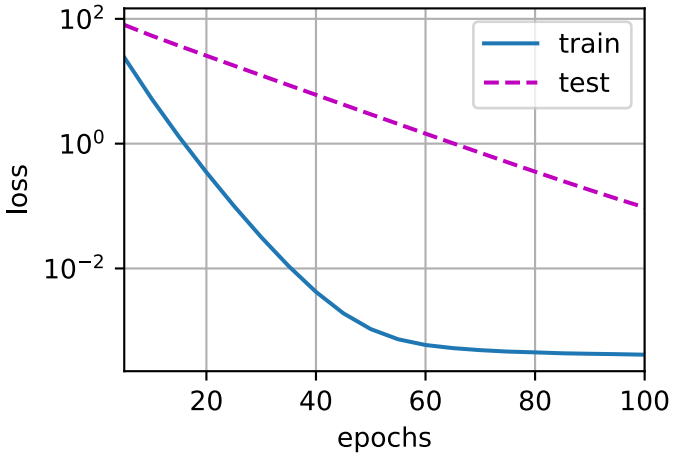


Using Weight Decay

Below, we run with substantial weight decay. Note that the training error increases but the test error decreases. This is precisely the effect we expect from regularization.

```
train(lambd=3)
```

```
L2 norm of w: 0.38248762
```



4.5.4 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.

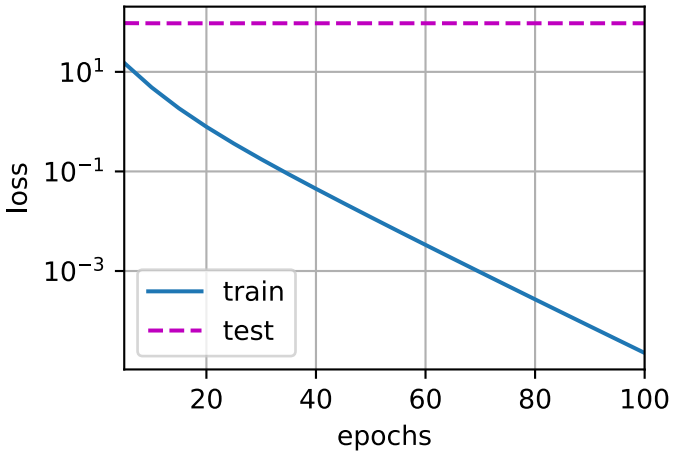
In the following code, we specify the weight decay hyperparameter directly through `wd` when instantiating our `Trainer`. By default, Gluon decays both weights and biases simultaneously. Note that the hyperparameter `wd` will be multiplied by `wd_mult` when updating model parameters. Thus, if we set `wd_mult` to zero, the bias parameter b will not decay.

```
def train_concise(wd):
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init.Normal(sigma=1))
    loss = gluon.loss.L2Loss()
    num_epochs, lr = 100, 0.003
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {
        'learning_rate': lr,
        'wd': wd})
    # The bias parameter has not decayed. Bias names generally end with "bias"
    net.collect_params('.*bias').setattr('wd_mult', 0)
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('L2 norm of w:', np.linalg.norm(net[0].weight.data()))
```

The plots look identical to those when we implemented weight decay from scratch. However, they run appreciably faster and are easier to implement, a benefit that will become more pronounced for larger problems.

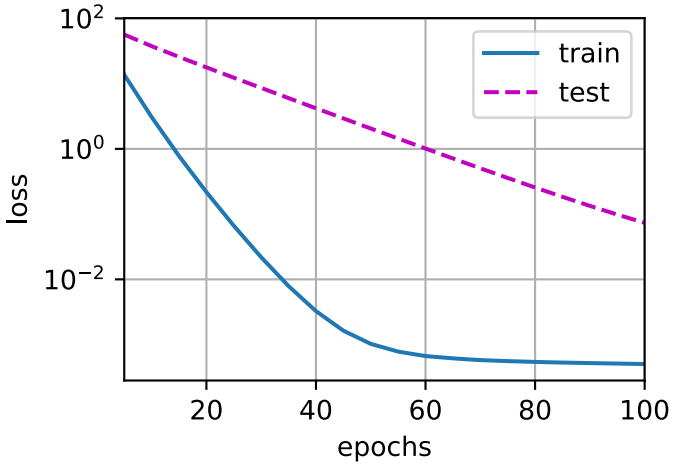
```
train_concise(0)
```

```
L2 norm of w: 15.014068
```




```
train_concise(3)
```

```
L2 norm of w: 0.33991417
```



So far, we only touched upon one notion of what constitutes a simple linear function. Moreover, what constitutes a simple nonlinear function can be an even more complex question. For instance, reproducing kernel Hilbert space (RKHS)[68] allows one to apply tools 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 default to the simple heuristic of applying weight decay on all layers of a deep network.

[68] https://en.wikipedia.org/wiki/Reproducing_kernel_Hilbert_space

Summary

- Regularization is a common method for dealing with overfitting. It adds a penalty term to the loss function on the training set to reduce the complexity of the learned model.
- One particular choice for keeping the model simple is weight decay using an L_2 penalty. This leads to weight decay in the update steps of the learning algorithm.
- 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.

Exercises

1. Experiment with the value of λ in the estimation problem in this section. Plot training and test 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 2.3.10)?
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 can you think of to 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[69]

4.6 Dropout

In Section 4.5, we introduced the classical approach to regularizing statistical models by penalizing the L_2 norm of the weights. In probabilistic terms, we could justify this technique by arguing that we have assumed a prior belief that weights take values from a Gaussian distribution with mean zero. More intuitively, we might argue that we encouraged the model to spread out its weights among many features rather than depending too much on a small number of potentially spurious associations.

[69] https://discuss.d2l.ai/t/98

4.6.1 Overfitting Revisited

Faced with more features than examples, linear models tend to overfit. But given more examples than features, we can generally count on linear models not to overfit. Unfortunately, the reliability with which linear models generalize comes at a cost. Naively applied, linear models do not take into account interactions among features. For every feature, a linear model must assign either a positive or a negative weight, ignoring context.

In traditional texts, this fundamental tension between generalizability and flexibility is described as the *bias-variance tradeoff*. Linear models have high bias: they can only represent a small class of functions. However, these models have low variance: they give similar results across different random samples of the data.

Deep neural networks inhabit the opposite end of the bias-variance spectrum. Unlike linear models, neural networks are not confined to looking at each feature individually. They can learn interactions among groups of features. For example, they might infer that "Nigeria" and "Western Union" appearing together in an email indicates spam but that separately they do not.

Even when we have far more examples than features, deep neural networks are capable of overfitting. In 2017, a group of researchers demonstrated the extreme flexibility of neural networks by training deep nets on randomly-labeled images. Despite the absence of any true pattern linking the inputs to the outputs, they found that the neural network optimized by stochastic gradient descent could label every image in the training set perfectly. Consider what this means. If the labels are assigned uniformly at random and there are 10 classes, then no classifier can do better than 10% accuracy on holdout data. The generalization gap here is a whopping 90%. If our models are so expressive that they can overfit this badly, then when should we expect them not to overfit?

The mathematical foundations for the puzzling generalization properties of deep networks remain open research questions, and we encourage the theoretically-oriented reader to dig deeper into the topic. For now, we turn to the investigation of practical tools that tend to empirically improve the generalization of deep nets.

4.6.2 Robustness through Perturbations

Let us think briefly about what we expect from a good predictive model. We want it to peform well on unseen data. Classical generalization theory suggests that to close the gap between train and test performance, we should aim for a simple model. Simplicity can come in the form of a small number of dimensions. We explored this when discussing the monomial basis functions of linear models in Section 4.4. Additionally, as we saw when discussing weight decay (L_2 regularization) in Section 4.5, the (inverse) norm of the parameters also represents a useful measure of simplicity. Another useful notion of simplicity is smoothness, i.e., that the function should not be sensitive to small changes to its inputs. For instance, when we classify images, we would expect that adding some random noise to the pixels should be mostly harmless.

In 1995, Christopher Bishop formalized this idea when he proved that training with input noise is equivalent to Tikhonov regularization (Bishop, 1995). This work drew a clear mathematical connection between the requirement that a function be smooth (and thus simple), and the requirement that it be resilient to perturbations in the input.

Then, in 2014, Srivastava et al. (Srivastava et al., 2014) developed a clever idea for how to apply Bishop's idea to the internal layers of a network, too. Namely, they proposed to inject noise into each layer of the network before calculating the subsequent layer during training. They realized

that when training a deep network with many layers, injecting noise enforces smoothness just on the input-output mapping.

Their idea, called *dropout*, involves injecting noise while computing each internal layer during forward propagation, and it has become a standard technique for training neural networks. The method is called *dropout* because we literally *drop out* some neurons during training. Throughout training, on each iteration, standard dropout consists of zeroing out some fraction of the nodes in each layer before calculating the subsequent layer.

To be clear, we are imposing our own narrative with the link to Bishop. The original paper on dropout offers intuition through a surprising analogy to sexual reproduction. The authors argue that neural network overfitting is characterized by a state in which each layer relies on a specifc pattern of activations in the previous layer, calling this condition *co-adaptation*. Dropout, they claim, breaks up co-adaptation just as sexual reproduction is argued to break up co-adapted genes.

The key challenge then is how to inject this noise. One idea is to inject the noise in an *unbiased* manner so that the expected value of each layer—while fixing the others—equals to the value it would have taken absent noise.

In Bishop's work, he added Gaussian noise to the inputs to a linear model. At each training iteration, he added noise sampled from a distribution with mean zero $\epsilon \sim \mathcal{N}(0, \sigma^2)$ to the input $\mathbf{x}$, yielding a perturbed point $\mathbf{x}' = \mathbf{x} + \epsilon$. In expectation, $E[\mathbf{x}'] = \mathbf{x}$.

In standard dropout regularization, one debiases each layer by normalizing by the fraction of nodes that were retained (not dropped out). In other words, with *dropout probability* p, each intermediate activation h is replaced by a random variable h' as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases} \tag{4.6.1}$$

By design, the expectation remains unchanged, i.e., $E[h'] = h$.

4.6.3 Dropout in Practice

Recall the MLP with a hidden layer and 5 hidden units in Fig. 4.1.1. When we apply dropout to a hidden layer, zeroing out each hidden unit with probability p, the result can be viewed as a network containing only a subset of the original neurons. In Fig. 4.6.1, h_2 and h_5 are removed. Consequently, the calculation of the outputs no longer depends on h_2 or h_5 and their respective gradient also vanishes when performing backpropagation. In this way, the calculation of the output layer cannot be overly dependent on any one element of $h_1, \ldots, h_5$.

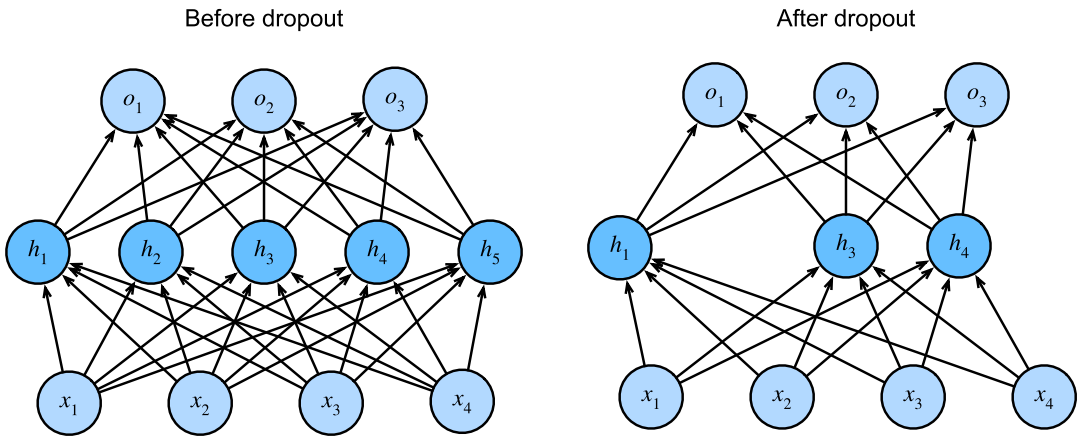


Fig. 4.6.1: MLP before and after dropout.

Typically, we disable dropout at test time. Given a trained model and a new example, we do not drop out any nodes and thus do not need to normalize. However, there are some exceptions: some researchers use dropout at test time as a heuristic for estimating the *uncertainty* of neural network predictions: if the predictions agree across many different dropout masks, then we might say that the network is more confident.

4.6.4 Implementation from Scratch

To implement the dropout function for a single layer, we must draw as many samples from a Bernoulli (binary) random variable as our layer has dimensions, where the random variable takes value 1 (keep) with probability $1-p$ and 0 (drop) with probability p. One easy way to implement this is to first draw samples from the uniform distribution $U[0,1]$. Then we can keep those nodes for which the corresponding sample is greater than p, dropping the rest.

In the following code, we implement a `dropout_layer` function that drops out the elements in the tensor input `X` with probability `dropout`, rescaling the remainder as described above: dividing the survivors by `1.0-dropout`.

```
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    # In this case, all elements are dropped out
    if dropout == 1:
        return np.zeros_like(X)
    # In this case, all elements are kept
    if dropout == 0:
        return X
    mask = np.random.uniform(0, 1, X.shape) > dropout
    return mask.astype(np.float32) * X / (1.0 - dropout)
```

We can test out the `dropout_layer` function on a few examples. In the following lines of code, we pass our input `X` through the dropout operation, with probabilities 0, 0.5, and 1, respectively.

```
X = np.arange(16).reshape(2, 8)
print(dropout_layer(X, 0))
print(dropout_layer(X, 0.5))
print(dropout_layer(X, 1))
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.]
 [ 8.  9. 10. 11. 12. 13. 14. 15.]]
[[ 0.  2.  4.  6.  8. 10. 12. 14.]
 [ 0. 18. 20.  0.  0.  0. 28.  0.]]
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

Defining Model Parameters

Again, we work with the Fashion-MNIST dataset introduced in Section 3.5. We define an MLP with two hidden layers containing 256 units each.

```
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256

W1 = np.random.normal(scale=0.01, size=(num_inputs, num_hiddens1))
b1 = np.zeros(num_hiddens1)
W2 = np.random.normal(scale=0.01, size=(num_hiddens1, num_hiddens2))
b2 = np.zeros(num_hiddens2)
W3 = np.random.normal(scale=0.01, size=(num_hiddens2, num_outputs))
b3 = np.zeros(num_outputs)

params = [W1, b1, W2, b2, W3, b3]
for param in params:
    param.attach_grad()
```

Defining the Model

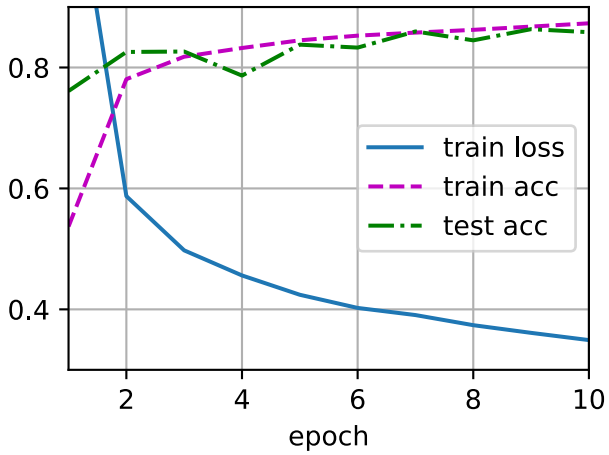
The model below applies dropout to the output of each hidden layer (following the activation function). We can set dropout probabilities for each layer separately. A common trend is to set a lower dropout probability closer to the input layer. Below we set it to 0.2 and 0.5 for the first and second hidden layers, respectively. We ensure that dropout is only active during training.

```
dropout1, dropout2 = 0.2, 0.5

def net(X):
    X = X.reshape(-1, num_inputs)
    H1 = npx.relu(np.dot(X, W1) + b1)
    # Use dropout only when training the model
    if autograd.is_training():
        # Add a dropout layer after the first fully connected layer
        H1 = dropout_layer(H1, dropout1)
    H2 = npx.relu(np.dot(H1, W2) + b2)
    if autograd.is_training():
        # Add a dropout layer after the second fully connected layer
        H2 = dropout_layer(H2, dropout2)
    return np.dot(H2, W3) + b3
```

Training and Testing

This is similar to the training and testing of MLPs described previously.

```
num_epochs, lr, batch_size = 10, 0.5, 256
loss = gluon.loss.SoftmaxCrossEntropyLoss()
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
              lambda batch_size: d2l.sgd(params, lr, batch_size))
```



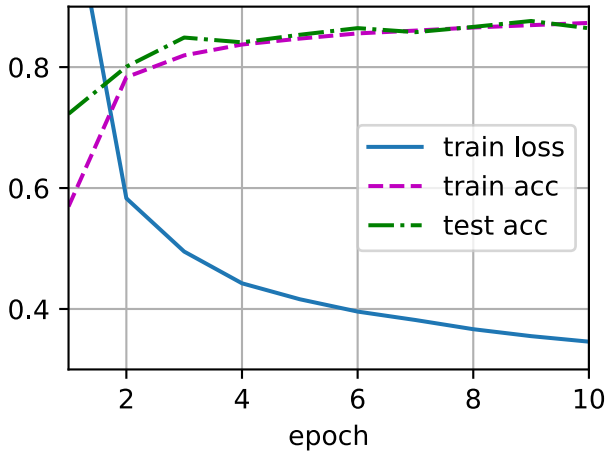
4.6.5 Concise Implementation

With high-level APIs, all we need to do is add a `Dropout` layer after each fully-connected layer, passing in the dropout probability as the only argument to its constructor. During training, the `Dropout` layer will randomly drop out outputs of the previous layer (or equivalently, the inputs to the subsequent layer) according to the specified dropout probability. When not in training mode, the `Dropout` layer simply passes the data through during testing.

```
net = nn.Sequential()
net.add(
    nn.Dense(256, activation="relu"),
    # Add a dropout layer after the first fully connected layer
    nn.Dropout(dropout1), nn.Dense(256, activation="relu"),
    # Add a dropout layer after the second fully connected layer
    nn.Dropout(dropout2), nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

Next, we train and test the model.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



Summary

- Beyond controlling the number of dimensions and the size of the weight vector, dropout is yet another tool to avoid overfitting. Often they are used jointly.
- Dropout replaces an activation h with a random variable with expected value h.
- Dropout is only used during training.

Exercises

1. What happens if you change the dropout probabilities for the first and second layers? In particular, what happens if you switch the ones for both layers? Design an experiment to answer these questions, describe your results quantitatively, and summarize the qualitative takeaways.
2. Increase the number of epochs and compare the results obtained when using dropout with those when not using it.
3. What is the variance of the activations in each hidden layer when dropout is and is not applied? Draw a plot to show how this quantity evolves over time for both models.
4. Why is dropout not typically used at test time?
5. Using the model in this section as an example, compare the effects of using dropout and weight decay. What happens when dropout and weight decay are used at the same time? Are the results additive? Are there diminished returns (or worse)? Do they cancel each other out?
6. What happens if we apply dropout to the individual weights of the weight matrix rather than the activations?
7. Invent another technique for injecting random noise at each layer that is different from the standard dropout technique. Can you develop a method that outperforms dropout on the Fashion-MNIST dataset (for a fixed architecture)?

Discussions[70]

[70] https://discuss.d2l.ai/t/100

4.7 Forward Propagation, Backward Propagation, and Computational Graphs

So far, we have trained our models with minibatch stochastic gradient descent. However, when we implemented the algorithm, we only worried about the calculations involved in *forward propagation* through the model. When it came time to calculate the gradients, we just invoked the backpropagation function provided by the deep learning framework.

The automatic calculation of gradients (automatic differentiation) profoundly simplifies the implementation of deep learning algorithms. Before automatic differentiation, even small changes to complicated models required recalculating complicated derivatives by hand. Surprisingly often, academic papers had to allocate numerous pages to deriving update rules. While we must continue to rely on automatic differentiation so we can focus on the interesting parts, you ought to know how these gradients are calculated under the hood if you want to go beyond a shallow understanding of deep learning.

In this section, we take a deep dive into the details of *backward propagation* (more commonly called *backpropagation*). To convey some insight for both the techniques and their implementations, we rely on some basic mathematics and computational graphs. To start, we focus our exposition on a one-hidden-layer MLP with weight decay (L_2 regularization).

4.7.1 Forward Propagation

Forward propagation (or *forward pass*) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer. We now work step-by-step through the mechanics of a neural network with one hidden layer. This may seem tedious but in the eternal words of funk virtuoso James Brown, you must "pay the cost to be the boss".

For the sake of simplicity, let us assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and that our hidden layer does not include a bias term. Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \tag{4.7.1}$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer. After running the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ through the activation function ϕ we obtain our hidden activation vector of length h,

$$\mathbf{h} = \phi(\mathbf{z}). \tag{4.7.2}$$

The hidden variable $\mathbf{h}$ is also an intermediate variable. Assuming that the parameters of the output layer only possess a weight of $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, we can obtain an output layer variable with a vector of length q:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \tag{4.7.3}$$

Assuming that the loss function is l and the example label is y, we can then calculate the loss term for a single data example,

$$L = l(\mathbf{o}, y). \tag{4.7.4}$$

According to the definition of L_2 regularization, given the hyperparameter λ, the regularization term is

$$s = \frac{\lambda}{2} \left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right), \tag{4.7.5}$$

where the Frobenius norm of the matrix is simply the L_2 norm applied after flattening the matrix into a vector. Finally, the model's regularized loss on a given data example is:

$$J = L + s. \tag{4.7.6}$$

We refer to J as the *objective function* in the following discussion.

4.7.2 Computational Graph of Forward Propagation

Plotting *computational graphs* helps us visualize the dependencies of operators and variables within the calculation. Fig. 4.7.1 contains the graph associated with the simple network described above, where squares denote variables and circles denote operators. The lower-left corner signifies the input and the upper-right corner is the output. Notice that the directions of the arrows (which illustrate data flow) are primarily rightward and upward.

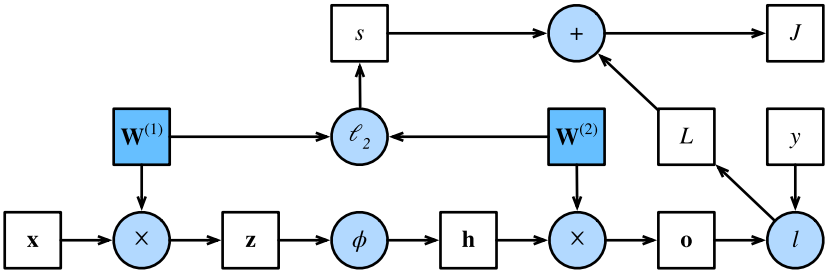


Fig. 4.7.1: Computational graph of forward propagation.

4.7.3 Backpropagation

Backpropagation refers to the method of calculating the gradient of neural network parameters. In short, the method traverses the network in reverse order, from the output to the input layer, according to the *chain rule* from calculus. The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters. Assume that we have functions $\mathsf{Y} = f(\mathsf{X})$ and $\mathsf{Z} = g(\mathsf{Y})$, in which the input and the output $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ are tensors of arbitrary shapes. By using the chain rule, we can compute the derivative of Z with respect to X via

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}} \right). \tag{4.7.7}$$

Here we use the prod operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions, have been carried out. For vectors, this is straightforward: it is simply matrix-matrix multiplication. For higher dimensional tensors, we use the appropriate counterpart. The operator prod hides all the notation overhead.

Recall that the parameters of the simple network with one hidden layer, whose computational graph is in Fig. 4.7.1, are $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. The objective of backpropagation is to calculate the gradients $\partial J/\partial \mathbf{W}^{(1)}$ and $\partial J/\partial \mathbf{W}^{(2)}$. To accomplish this, we apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter. The order of calculations are reversed

relative to those performed in forward propagation, since we need to start with the outcome of the computational graph and work our way towards the parameters. The first step is to calculate the gradients of the objective function $J = L + s$ with respect to the loss term L and the regularization term s.

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1. \tag{4.7.8}$$

Next, we compute the gradient of the objective function with respect to variable of the output layer $\mathbf{o}$ according to the chain rule:

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q. \tag{4.7.9}$$

Next, we calculate the gradients of the regularization term with respect to both parameters:

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}. \tag{4.7.10}$$

Now we are able to calculate the gradient $\partial J / \partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ of the model parameters closest to the output layer. Using the chain rule yields:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}. \tag{4.7.11}$$

To obtain the gradient with respect to $\mathbf{W}^{(1)}$ we need to continue backpropagation along the output layer to the hidden layer. The gradient with respect to the hidden layer's outputs $\partial J / \partial \mathbf{h} \in \mathbb{R}^h$ is given by

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}. \tag{4.7.12}$$

Since the activation function ϕ applies elementwise, calculating the gradient $\partial J / \partial \mathbf{z} \in \mathbb{R}^h$ of the intermediate variable $\mathbf{z}$ requires that we use the elementwise multiplication operator, which we denote by $\odot$:

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right). \tag{4.7.13}$$

Finally, we can obtain the gradient $\partial J / \partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ of the model parameters closest to the input layer. According to the chain rule, we get

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \tag{4.7.14}$$

4.7.4 Training Neural Networks

When training neural networks, forward and backward propagation depend on each other. In particular, for forward propagation, we traverse the computational graph in the direction of dependencies and compute all the variables on its path. These are then used for backpropagation where the compute order on the graph is reversed.

Take the aforementioned simple network as an example to illustrate. On one hand, computing the regularization term (4.7.5) during forward propagation depends on the current values of model

parameters $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. They are given by the optimization algorithm according to backpropagation in the latest iteration. On the other hand, the gradient calculation for the parameter (4.7.11) during backpropagation depends on the current value of the hidden variable $\mathbf{h}$, which is given by forward propagation.

Therefore when training neural networks, after model parameters are initialized, we alternate forward propagation with backpropagation, updating model parameters using gradients given by backpropagation. Note that backpropagation reuses the stored intermediate values from forward propagation to avoid duplicate calculations. One of the consequences is that we need to retain the intermediate values until backpropagation is complete. This is also one of the reasons why training requires significantly more memory than plain prediction. Besides, the size of such intermediate values is roughly proportional to the number of network layers and the batch size. Thus, training deeper networks using larger batch sizes more easily leads to *out of memory* errors.

Summary

- Forward propagation sequentially calculates and stores intermediate variables within the computational graph defined by the neural network. It proceeds from the input to the output layer.
- Backpropagation sequentially calculates and stores the gradients of intermediate variables and parameters within the neural network in the reversed order.
- When training deep learning models, forward propagation and back propagation are interdependent.
- Training requires significantly more memory than prediction.

Exercises

1. Assume that the inputs $\mathbf{X}$ to some scalar function f are $n \times m$ matrices. What is the dimensionality of the gradient of f with respect to $\mathbf{X}$?
2. Add a bias to the hidden layer of the model described in this section (you do not need to include bias in the regularization term).
    1. Draw the corresponding computational graph.
    2. Derive the forward and backward propagation equations.
3. Compute the memory footprint for training and prediction in the model described in this section.
4. Assume that you want to compute second derivatives. What happens to the computational graph? How long do you expect the calculation to take?
5. Assume that the computational graph is too large for your GPU.
    1. Can you partition it over more than one GPU?
    2. What are the advantages and disadvantages over training on a smaller minibatch?

Discussions[71]

[71] https://discuss.d2l.ai/t/102

4.8 Numerical Stability and Initialization

Thus far, every model that we have implemented required that we initialize its parameters according to some pre-specified distribution. Until now, we took the initialization scheme for granted, glossing over the details of how these choices are made. You might have even gotten the impression that these choices are not especially important. To the contrary, the choice of initialization scheme plays a significant role in neural network learning, and it can be crucial for maintaining numerical stability. Moreover, these choices can be tied up in interesting ways with the choice of the nonlinear activation function. Which function we choose and how we initialize parameters can determine how quickly our optimization algorithm converges. Poor choices here can cause us to encounter exploding or vanishing gradients while training. In this section, we delve into these topics with greater detail and discuss some useful heuristics that you will find useful throughout your career in deep learning.

4.8.1 Vanishing and Exploding Gradients

Consider a deep network with L layers, input $\mathbf{x}$ and output $\mathbf{o}$. With each layer l defined by a transformation f_l parameterized by weights $\mathbf{W}^{(l)}$, whose hidden variable is $\mathbf{h}^{(l)}$ (let $\mathbf{h}^{(0)} = \mathbf{x}$), our network can be expressed as:

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \ldots \circ f_1(\mathbf{x}). \tag{4.8.1}$$

If all the hidden variables and the input are vectors, we can write the gradient of $\mathbf{o}$ with respect to any set of parameters $\mathbf{W}^{(l)}$ as follows:

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}. \tag{4.8.2}$$

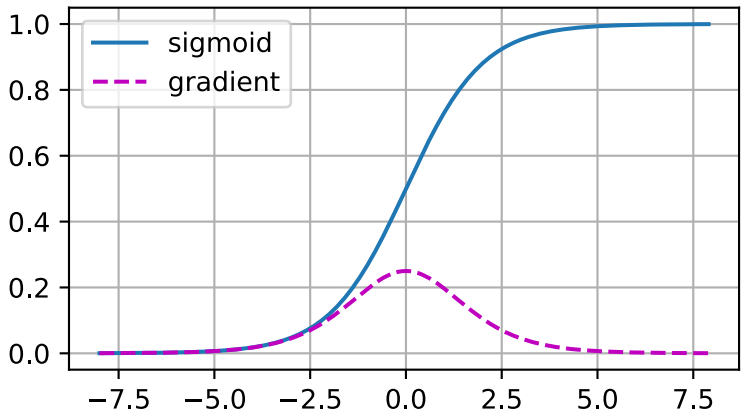
In other words, this gradient is the product of $L-l$ matrices $\mathbf{M}^{(L)} \cdot \ldots \cdot \mathbf{M}^{(l+1)}$ and the gradient vector $\mathbf{v}^{(l)}$. Thus we are susceptible to the same problems of numerical underflow that often crop up when multiplying together too many probabilities. When dealing with probabilities, a common trick is to switch into log-space, i.e., shifting pressure from the mantissa to the exponent of the numerical representation. Unfortunately, our problem above is more serious: initially the matrices $\mathbf{M}^{(l)}$ may have a wide variety of eigenvalues. They might be small or large, and their product might be *very large* or *very small*.

The risks posed by unstable gradients go beyond numerical representation. Gradients of unpredictable magnitude also threaten the stability of our optimization algorithms. We may be facing parameter updates that are either (i) excessively large, destroying our model (the *exploding gradient* problem); or (ii) excessively small (the *vanishing gradient* problem), rendering learning impossible as parameters hardly move on each update.

Vanishing Gradients

One frequent culprit causing the vanishing gradient problem is the choice of the activation function σ that is appended following each layer's linear operations. Historically, the sigmoid function $1/(1+\exp(-x))$ (introduced in Section 4.1) was popular because it resembles a thresholding function. Since early artificial neural networks were inspired by biological neural networks, the idea of neurons that fire either *fully* or *not at all* (like biological neurons) seemed appealing. Let us take a closer look at the sigmoid to see why it can cause vanishing gradients.

```
%matplotlib inline
from mxnet import autograd, np, npx
from d2l import mxnet as d2l

npx.set_np()

x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.sigmoid(x)
y.backward()

d2l.plot(x, [y, x.grad], legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



As you can see, the sigmoid's gradient vanishes both when its inputs are large and when they are small. Moreover, when backpropagating through many layers, unless we are in the Goldilocks zone, where the inputs to many of the sigmoids are close to zero, the gradients of the overall product may vanish. When our network boasts many layers, unless we are careful, the gradient will likely be cut off at some layer. Indeed, this problem used to plague deep network training. Consequently, ReLUs, which are more stable (but less neurally plausible), have emerged as the default choice for practitioners.

Exploding Gradients

The opposite problem, when gradients explode, can be similarly vexing. To illustrate this a bit better, we draw 100 Gaussian random matrices and multiply them with some initial matrix. For the scale that we picked (the choice of the variance $\sigma^2 = 1$), the matrix product explodes. When this happens due to the initialization of a deep network, we have no chance of getting a gradient descent optimizer to converge.

```
M = np.random.normal(size=(4, 4))
print('a single matrix', M)
for i in range(100):
    M = np.dot(M, np.random.normal(size=(4, 4)))

print('after multiplying 100 matrices', M)
```

```
a single matrix [[ 2.2122064   1.1630787   0.7740038   0.4838046 ]
 [ 1.0434405   0.29956347  1.1839255   0.15302546]
 [ 1.8917114  -1.1688148  -1.2347414   1.5580711 ]
 [-1.771029   -0.5459446  -0.45138445 -2.3556297 ]]
after multiplying 100 matrices [[ 3.4459714e+23 -7.8040680e+23  5.9973287e+23  4.5229990e+23]
 [ 2.5275089e+23 -5.7240326e+23  4.3988473e+23  3.3174740e+23]
 [ 1.3731286e+24 -3.1097155e+24  2.3897773e+24  1.8022959e+24]
 [-4.4951040e+23  1.0180033e+24 -7.8232281e+23 -5.9000354e+23]]
```

Breaking the Symmetry

Another problem in neural network design is the symmetry inherent in their parametrization. Assume that we have a simple MLP with one hidden layer and two units. In this case, we could permute the weights $\mathbf{W}^{(1)}$ of the first layer and likewise permute the weights of the output layer to obtain the same function. There is nothing special differentiating the first hidden unit vs. the second hidden unit. In other words, we have permutation symmetry among the hidden units of each layer.

This is more than just a theoretical nuisance. Consider the aforementioned one-hidden-layer MLP with two hidden units. For illustration, suppose that the output layer transforms the two hidden units into only one output unit. Imagine what would happen if we initialized all of the parameters of the hidden layer as $\mathbf{W}^{(1)} = c$ for some constant c. In this case, during forward propagation either hidden unit takes the same inputs and parameters, producing the same activation, which is fed to the output unit. During backpropagation, differentiating the output unit with respect to parameters $\mathbf{W}^{(1)}$ gives a gradient whose elements all take the same value. Thus, after gradient-based iteration (e.g., minibatch stochastic gradient descent), all the elements of $\mathbf{W}^{(1)}$ still take the same value. Such iterations would never *break the symmetry* on its own and we might never be able to realize the network's expressive power. The hidden layer would behave as if it had only a single unit. Note that while minibatch stochastic gradient descent would not break this symmetry, dropout regularization would!

4.8.2 Parameter Initialization

One way of addressing—or at least mitigating—the issues raised above is through careful initialization. Additional care during optimization and suitable regularization can further enhance stability.

Default Initialization

In the previous sections, e.g., in Section 3.3, we used a normal distribution to initialize the values of our weights. If we do not specify the initialization method, the framework will use a default random initialization method, which often works well in practice for moderate problem sizes.

Xavier Initialization

Let us look at the scale distribution of an output (e.g., a hidden variable) o_i for some fully-connected layer *without nonlinearities*. With n_{in} inputs x_j and their associated weights w_{ij} for this layer, an output is given by

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j. \tag{4.8.3}$$

The weights w_{ij} are all drawn independently from the same distribution. Furthermore, let us assume that this distribution has zero mean and variance σ^2. Note that this does not mean that the distribution has to be Gaussian, just that the mean and variance need to exist. For now, let us assume that the inputs to the layer x_j also have zero mean and variance γ^2 and that they are independent of w_{ij} and independent of each other. In this case, we can compute the mean and variance of o_i as follows:

$$\begin{aligned}
E[o_i] &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j] \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}] E[x_j] \\
&= 0, \\
\text{Var}[o_i] &= E[o_i^2] - (E[o_i])^2 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2] E[x_j^2] \\
&= n_{\text{in}} \sigma^2 \gamma^2.
\end{aligned} \tag{4.8.4}$$

One way to keep the variance fixed is to set $n_{\text{in}} \sigma^2 = 1$. Now consider backpropagation. There we face a similar problem, albeit with gradients being propagated from the layers closer to the output. Using the same reasoning as for forward propagation, we see that the gradients' variance can blow up unless $n_{\text{out}} \sigma^2 = 1$, where n_{out} is the number of outputs of this layer. This leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously. Instead, we simply try to

satisfy:

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}. \tag{4.8.5}$$

This is the reasoning underlying the now-standard and practically beneficial *Xavier initialization*, named after the first author of its creators (Glorot & Bengio, 2010). Typically, the Xavier initialization samples weights from a Gaussian distribution with zero mean and variance $\sigma^2 = \frac{2}{n_{\text{in}}+n_{\text{out}}}$. We can also adapt Xavier's intuition to choose the variance when sampling weights from a uniform distribution. Note that the uniform distribution $U(-a, a)$ has variance $\frac{a^2}{3}$. Plugging $\frac{a^2}{3}$ into our condition on σ^2 yields the suggestion to initialize according to

$$U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right). \tag{4.8.6}$$

Though the assumption for nonexistence of nonlinearities in the above mathematical reasoning can be easily violated in neural networks, the Xavier initialization method turns out to work well in practice.

Beyond

The reasoning above barely scratches the surface of modern approaches to parameter initialization. A deep learning framework often implements over a dozen different heuristics. Moreover, parameter initialization continues to be a hot area of fundamental research in deep learning. Among these are heuristics specialized for tied (shared) parameters, super-resolution, sequence models, and other situations. For instance, Xiao et al. demonstrated the possibility of training 10000-layer neural networks without architectural tricks by using a carefully-designed initialization method (Xiao et al., 2018).

If the topic interests you we suggest a deep dive into this module's offerings, reading the papers that proposed and analyzed each heuristic, and then exploring the latest publications on the topic. Perhaps you will stumble across or even invent a clever idea and contribute an implementation to deep learning frameworks.

Summary

- Vanishing and exploding gradients are common issues in deep networks. Great care in parameter initialization is required to ensure that gradients and parameters remain well controlled.
- Initialization heuristics are needed to ensure that the initial gradients are neither too large nor too small.
- ReLU activation functions mitigate the vanishing gradient problem. This can accelerate convergence.
- Random initialization is key to ensure that symmetry is broken before optimization.
- Xavier initialization suggests that, for each layer, variance of any output is not affected by the number of inputs, and variance of any gradient is not affected by the number of outputs.

Exercises

1. Can you design other cases where a neural network might exhibit symmetry requiring breaking besides the permutation symmetry in an MLP's layers?
2. Can we initialize all weight parameters in linear regression or in softmax regression to the same value?
3. Look up analytic bounds on the eigenvalues of the product of two matrices. What does this tell you about ensuring that gradients are well conditioned?
4. If we know that some terms diverge, can we fix this after the fact? Look at the paper on layerwise adaptive rate scaling for inspiration (You et al., 2017).

Discussions[72]

4.9 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 come from in the first place or what we plan to ultimately 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 pattern. 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 vs. 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 to thereafter grant loans to all applicants 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 coinciding 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 these 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.

[72] https://discuss.d2l.ai/t/103

4.9.1 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 were 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: $p_S(y|\mathbf{x}) = 1 - p_T(y|\mathbf{x})$. 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. 4.9.1.

Fig. 4.9.1: Training data for distinguishing cats and dogs.

At test time we are asked to classify the images in Fig. 4.9.2.

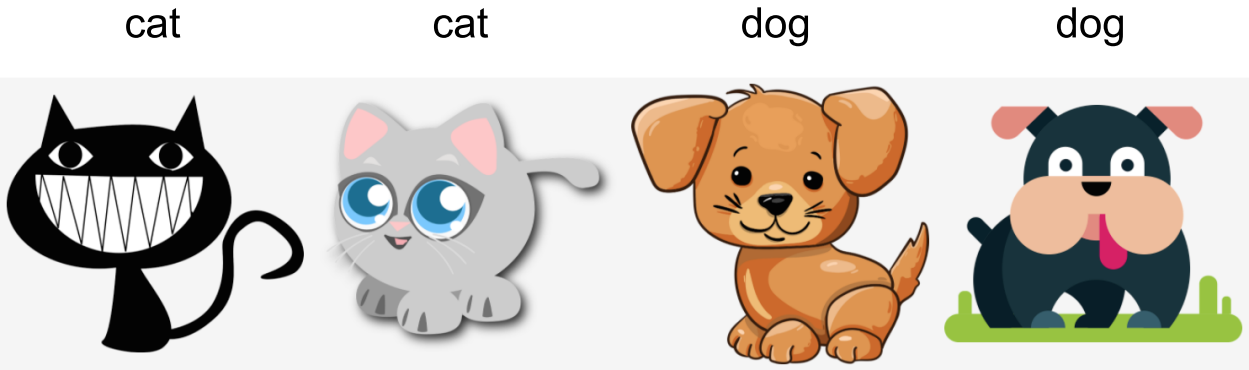


Fig. 4.9.2: Test data for distinguishing cats and dogs.

The training set consists of photos, while the test set contains only cartoons. Training on a 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. 4.9.3.

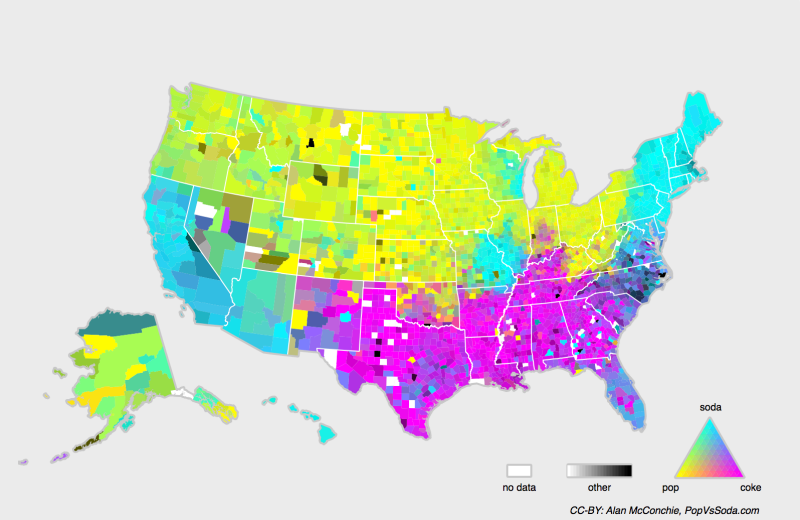


Fig. 4.9.3: Concept shift on soft drink names in the United States.

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.

4.9.2 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 us (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 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 are 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 wisen 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 were 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.

4.9.3 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 us first reflect about 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:

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(f(\mathbf{x}_i), y_i), \tag{4.9.1}$$

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 (4.9.1) *empirical risk*. The *empirical risk* is an average loss over the training data to approximate the *risk*, which is the expectation of the loss over the entire population of data drawn from their true distribution $p(\mathbf{x}, y)$:

$$E_{p(\mathbf{x},y)}[l(f(\mathbf{x}), y)] = \int\int l(f(\mathbf{x}), y) p(\mathbf{x}, y) \, d\mathbf{x}dy. \tag{4.9.2}$$

However, in practice we typically cannot obtain the entire population of data. Thus, *empirical risk minimization*, which is minimizing the empirical risk in (4.9.1), is a practical strategy for machine learning, with the hope to approximate minimizing the risk.

Covariate Shift Correction

Assume that we want to estimate some dependency $P(y \mid \mathbf{x})$ for which we have labeled data $(\mathbf{x}_i, y_i)$. Unfortunately, the observations $\mathbf{x}_i$ are drawn from some *source distribution* $q(\mathbf{x})$ rather than the *target distribution* $p(\mathbf{x})$. Fortunately, the dependency assumption means that the conditional distribution does not change: $p(y \mid \mathbf{x}) = q(y \mid \mathbf{x})$. If the source distribution $q(\mathbf{x})$ is "wrong", we can correct for that by using the following simple identity in the risk:

$$\int\int l(f(\mathbf{x}), y) p(y \mid \mathbf{x}) p(\mathbf{x}) \, d\mathbf{x}dy = \int\int l(f(\mathbf{x}), y) q(y \mid \mathbf{x}) q(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} \, d\mathbf{x}dy. \tag{4.9.3}$$

In other words, we need to reweigh each data example by the ratio of the probability that it would have been drawn from the correct distribution to that from the wrong one:

$$\beta_i \stackrel{\text{def}}{=} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)}. \tag{4.9.4}$$

Plugging in the weight β_i for each data example $(\mathbf{x}_i, y_i)$ we can train our model using *weighted empirical risk minimization*:

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} \beta_i l(f(\mathbf{x}_i), y_i). \tag{4.9.5}$$

Alas, we do not know that ratio, so before we can do anything useful we need to estimate it. Many methods are available, including some fancy operator-theoretic approaches that attempt to recalibrate the expectation operator directly using a minimum-norm or a maximum entropy principle. Note that for any such approach, we need samples drawn from both distributions—the "true" p, e.g., by access to test data, and the one used for generating the training set q (the latter is trivially available). Note however, that we only need features $\mathbf{x} \sim p(\mathbf{x})$; we do not need to access labels $y \sim p(y)$.

In this case, there exists a very effective approach that will give almost as good results as the original: logistic regression, which is a special case of softmax regression (see Section 3.4) for binary classification. This is all that is needed to compute estimated probability ratios. We learn a classifier to distinguish between data drawn from $p(\mathbf{x})$ and data drawn from $q(\mathbf{x})$. If it is impossible to distinguish between the two distributions then it means that the associated instances are equally likely to come from either one of the two distributions. On the other hand, any instances that can be well discriminated should be significantly overweighted or underweighted accordingly.

For simplicity's sake assume that we have an equal number of instances from both distributions $p(\mathbf{x})$ and $q(\mathbf{x})$, respectively. Now denote by z labels that are 1 for data drawn from p and -1 for data drawn from q. Then the probability in a mixed dataset is given by

$$P(z = 1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x}) + q(\mathbf{x})} \text{ and hence } \frac{P(z = 1 \mid \mathbf{x})}{P(z = -1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}. \tag{4.9.6}$$

Thus, if we use a logistic regression approach, where $P(z = 1 \mid \mathbf{x}) = \frac{1}{1+\exp(-h(\mathbf{x}))}$ (h is a parameterized function), it follows that

$$\beta_i = \frac{1/(1 + \exp(-h(\mathbf{x}_i)))}{\exp(-h(\mathbf{x}_i))/(1 + \exp(-h(\mathbf{x}_i)))} = \exp(h(\mathbf{x}_i)). \tag{4.9.7}$$

As a result, we need to solve two problems: first one to distinguish between data drawn from both distributions, and then a weighted empirical risk minimization problem in (4.9.5) where we weigh terms by β_i.

Now we are ready to describe a correction algorithm. Suppose that we have a training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and an unlabeled test set $\{\mathbf{u}_1, \ldots, \mathbf{u}_m\}$. For covariate shift, we assume that $\mathbf{x}_i$ for all $1 \leq i \leq n$ are drawn from some source distribution and $\mathbf{u}_i$ for all $1 \leq i \leq m$ are drawn from the target distribution. Here is a prototypical algorithm for correcting covariate shift:

1. Generate a binary-classification training set: $\{(\mathbf{x}_1, -1), \ldots, (\mathbf{x}_n, -1), (\mathbf{u}_1, 1), \ldots, (\mathbf{u}_m, 1)\}$.
2. Train a binary classifier using logistic regression to get function h.
3. Weigh training data using $\beta_i = \exp(h(\mathbf{x}_i))$ or better $\beta_i = \min(\exp(h(\mathbf{x}_i)), c)$ for some constant c.
4. Use weights β_i for training on $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ in (4.9.5).

Note that the above algorithm relies on a crucial assumption. For this scheme to work, we need that each data example in the target (e.g., test time) distribution had nonzero probability of occurring at training time. If we find a point where $p(\mathbf{x}) > 0$ but $q(\mathbf{x}) = 0$, then the corresponding importance weight should be infinity.

Label Shift Correction

Assume that we are dealing with a classification task with k categories. Using the same notation in Section 4.9.3, q and p are the source distribution (e.g., training time) and target distribution (e.g., test time), respectively. Assume that the distribution of labels shifts over time: $q(y) \neq p(y)$, but the class-conditional distribution stays the same: $q(\mathbf{x} \mid y) = p(\mathbf{x} \mid y)$. If the source distribution $q(y)$ is "wrong", we can correct for that according to the following identity in the risk as defined in (4.9.2):

$$\int \int l(f(\mathbf{x}), y) p(\mathbf{x} \mid y) p(y) \, d\mathbf{x} dy = \int \int l(f(\mathbf{x}), y) q(\mathbf{x} \mid y) q(y) \frac{p(y)}{q(y)} \, d\mathbf{x} dy. \tag{4.9.8}$$

Here, our importance weights will correspond to the label likelihood ratios

$$\beta_i \stackrel{\text{def}}{=} \frac{p(y_i)}{q(y_i)}. \tag{4.9.9}$$

One nice thing about label shift is that if we have a reasonably good model on the source distribution, then we can get consistent estimates of these weights without ever having to deal with the ambient dimension. In deep learning, the inputs tend to be high-dimensional objects like images, while the labels are often simpler objects like categories.

To estimate the target label distribution, we first take our reasonably good off-the-shelf classifier (typically trained on the training data) and compute its confusion matrix using the validation set (also from the training distribution). The *confusion matrix*, $\mathbf{C}$, is simply a $k \times k$ matrix, where each column corresponds to the label category (ground truth) and each row corresponds to our model's predicted category. Each cell's value c_{ij} is the fraction of total predictions on the validation set where the true label was j and our model predicted i.

Now, we cannot calculate the confusion matrix on the target data directly, because we do not get to see the labels for the examples that we see in the wild, unless we invest in a complex real-time annotation pipeline. What we can do, however, is average all of our models predictions at test time together, yielding the mean model outputs $\mu(\hat{\mathbf{y}}) \in \mathbb{R}^k$, whose i^{th} element $\mu(\hat{y}_i)$ is the fraction of total predictions on the test set where our model predicted i.

It turns out that under some mild conditions—if our classifier was reasonably accurate in the first place, and if the target data contain only categories that we have seen before, and if the label shift assumption holds in the first place (the strongest assumption here), then we can estimate the test set label distribution by solving a simple linear system

$$\mathbf{C} p(\mathbf{y}) = \mu(\hat{\mathbf{y}}), \tag{4.9.10}$$

because as an estimate $\sum_{j=1}^{k} c_{ij} p(y_j) = \mu(\hat{y}_i)$ holds for all $1 \leq i \leq k$, where $p(y_j)$ is the j^{th} element of the k-dimensional label distribution vector $p(\mathbf{y})$. If our classifier is sufficiently accurate to begin with, then the confusion matrix $\mathbf{C}$ will be invertible, and we get a solution $p(\mathbf{y}) = \mathbf{C}^{-1} \mu(\hat{\mathbf{y}})$.

Because we observe the labels on the source data, it is easy to estimate the distribution $q(y)$. Then for any training example i with label y_i, we can take the ratio of our estimated $p(y_i)/q(y_i)$ to calculate the weight β_i, and plug this into weighted empirical risk minimization in (4.9.5).

Concept Shift Correction

Concept shift is much harder to fix in a principled manner. For instance, in a situation where suddenly the problem changes from distinguishing cats from dogs to one of distinguishing white from black animals, it will be unreasonable to assume that we can do much better than just collecting new labels and training from scratch. Fortunately, in practice, such extreme shifts are rare. Instead, what usually happens is that the task keeps on changing slowly. To make things more concrete, here are some examples:

- In computational advertising, new products are launched, old products become less popular. This means that the distribution over ads and their popularity changes gradually and any click-through rate predictor needs to change gradually with it.
- Traffic camera lenses degrade gradually due to environmental wear, affecting image quality progressively.
- News content changes gradually (i.e., most of the news remains unchanged but new stories appear).

In such cases, we can use the same approach that we used for training networks to make them adapt to the change in the data. In other words, we use the existing network weights and simply perform a few update steps with the new data rather than training from scratch.

4.9.4 A Taxonomy of Learning Problems

Armed with knowledge about how to deal with changes in distributions, we can now consider some other aspects of machine learning problem formulation.

Batch Learning

In *batch learning*, we have access to training features and labels $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, which we use to train a model $f(\mathbf{x})$. Later on, we deploy this model to score new data $(\mathbf{x}, y)$ drawn from the same distribution. This is the default assumption for any of the problems that we discuss here. For instance, we might train a cat detector based on lots of pictures of cats and dogs. Once we trained it, we ship it as part of a smart catdoor computer vision system that lets only cats in. This is then installed in a customer's home and is never updated again (barring extreme circumstances).

Online Learning

Now imagine that the data $(\mathbf{x}_i, y_i)$ arrives one sample at a time. More specifically, assume that we first observe $\mathbf{x}_i$, then we need to come up with an estimate $f(\mathbf{x}_i)$ and only once we have done this, we observe y_i and with it, we receive a reward or incur a loss, given our decision. Many real problems fall into this category. For example, we need to predict tomorrow's stock price, this allows us to trade based on that estimate and at the end of the day we find out whether our estimate allowed us to make a profit. In other words, in *online learning*, we have the following cycle where we are continuously improving our model given new observations.

$$\text{model } f_t \longrightarrow \text{data } \mathbf{x}_t \longrightarrow \text{estimate } f_t(\mathbf{x}_t) \longrightarrow \text{observation } y_t \longrightarrow \text{loss } l(y_t, f_t(\mathbf{x}_t)) \longrightarrow \text{model } f_{t+1} \tag{4.9.11}$$

Bandits

Bandits are a special case of the problem above. While in most learning problems we have a continuously parametrized function f where we want to learn its parameters (e.g., a deep network), in a *bandit* problem we only have a finite number of arms that we can pull, i.e., a finite number of actions that we can take. It is not very surprising that for this simpler problem stronger theoretical guarantees in terms of optimality can be obtained. We list it mainly since this problem is often (confusingly) treated as if it were a distinct learning setting.

Control

In many cases the environment remembers what we did. Not necessarily in an adversarial manner but it will just remember and the response will depend on what happened before. For instance, a coffee boiler controller will observe different temperatures depending on whether it was heating the boiler previously. PID (proportional-integral-derivative) controller algorithms are a popular choice there. Likewise, a user's behavior on a news site will depend on what we showed him previously (e.g., he will read most news only once). Many such algorithms form a model of the environment in which they act such as to make their decisions appear less random. Recently, control theory (e.g., PID variants) has also been used to automatically tune hyperparameters to achive better disentangling and reconstruction quality, and improve the diversity of generated text and the reconstruction quality of generated images (Shao et al., 2020).

Reinforcement Learning

In the more general case of an environment with memory, we may encounter situations where the environment is trying to cooperate with us (cooperative games, in particular for non-zero-sum games), or others where the environment will try to win. Chess, Go, Backgammon, or StarCraft are some of the cases in *reinforcement learning*. Likewise, we might want to build a good controller for autonomous cars. The other cars are likely to respond to the autonomous car's driving style in nontrivial ways, e.g., trying to avoid it, trying to cause an accident, and trying to cooperate with it.

Considering the Environment

One key distinction between the different situations above is that the same strategy that might have worked throughout in the case of a stationary environment, might not work throughout when the environment can adapt. For instance, an arbitrage opportunity discovered by a trader is likely to disappear once he starts exploiting it. The speed and manner at which the environment changes determines to a large extent the type of algorithms that we can bring to bear. For instance, if we know that things may only change slowly, we can force any estimate to change only slowly, too. If we know that the environment might change instantaneously, but only very infrequently, we can make allowances for that. These types of knowledge are crucial for the aspiring data scientist to deal with concept shift, i.e., when the problem that he is trying to solve changes over time.

4.9.5 Fairness, Accountability, and Transparency in Machine Learning

Finally, it is important to remember that when you deploy machine learning systems you are not merely optimizing a predictive model—you are typically providing a tool that will be used to (partially or fully) automate decisions. These technical systems can impact the lives of individuals subject to the resulting decisions. The leap from considering predictions to decisions raises not only new technical questions, but also a slew of ethical questions that must be carefully considered. If we are deploying a medical diagnostic system, we need to know for which populations it may work and which it may not. Overlooking foreseeable risks to the welfare of a subpopulation could cause us to administer inferior care. Moreover, once we contemplate decision-making systems, we must step back and reconsider how we evaluate our technology. Among other consequences of this change of scope, we will find that *accuracy* is seldom the right measure. For instance, when translating predictions into actions, we will often want to take into account the potential cost sensitivity of erring in various ways. If one way of misclassifying an image could be perceived as a racial sleight of hand, while misclassification to a different category would be harmless, then we might want to adjust our thresholds accordingly, accounting for societal values in designing the decision-making protocol. We also want to be careful about how prediction systems can lead to feedback loops. For example, consider predictive policing systems, which allocate patrol officers to areas with high forecasted crime. It is easy to see how a worrying pattern can emerge:

1. Neighborhoods with more crime get more patrols.
2. Consequently, more crimes are discovered in these neighborhoods, entering the training data available for future iterations.
3. Exposed to more positives, the model predicts yet more crime in these neighborhoods.
4. In the next iteration, the updated model targets the same neighborhood even more heavily leading to yet more crimes discovered, etc.

Often, the various mechanisms by which a model's predictions become coupled to its training data are unaccounted for in the modeling process. This can lead to what researchers call *runaway feedback loops*. Additionally, we want to be careful about whether we are addressing the right problem in the first place. Predictive algorithms now play an outsize role in mediating the dissemination of information. Should the news that an individual encounters be determined by the set of Facebook pages they have *Liked*? These are just a few among the many pressing ethical dilemmas that you might encounter in a career in machine learning.

Summary

- In many cases training and test sets do not come from the same distribution. This is called distribution shift.
- The risk is the expectation of the loss over the entire population of data drawn from their true distribution. However, this entire population is usually unavailable. Empirical risk is an average loss over the training data to approximate the risk. In practice, we perform empirical risk minimization.
- Under the corresponding assumptions, covariate and label shift can be detected and corrected for at test time. Failure to account for this bias can become problematic at test time.
- In some cases, the environment may remember automated actions and respond in surprising ways. We must account for this possibility when building models and continue to mon-

itor live systems, open to the possibility that our models and the environment will become entangled in unanticipated ways.

Exercises

1. What could happen when we change the behavior of a search engine? What might the users do? What about the advertisers?
2. Implement a covariate shift detector. Hint: build a classifier.
3. Implement a covariate shift corrector.
4. Besides distribution shift, what else could affect how the empirical risk approximates the risk?

Discussions[73]

4.10 Predicting House Prices on Kaggle

Now that we have introduced some basic tools for building and training deep networks and regularizing them with techniques including weight decay and dropout, we are ready to put all this knowledge into practice by participating in a Kaggle competition. The house price prediction competition is a great place to start. The data are fairly generic and do not exhibit exotic structure that might require specialized models (as audio or video might). This dataset, collected by Bart de Cock in 2011 (DeCock, 2011), covers house prices in Ames, IA from the period of 2006–2010. It is considerably larger than the famous Boston housing dataset[74] of Harrison and Rubinfeld (1978), boasting both more examples and more features.

In this section, we will walk you through details of data preprocessing, model design, and hyperparameter selection. We hope that through a hands-on approach, you will gain some intuitions that will guide you in your career as a data scientist.

4.10.1 Downloading and Caching Datasets

Throughout the book, we will train and test models on various downloaded datasets. Here, we implement several utility functions to facilitate data downloading. First, we maintain a dictionary `DATA_HUB` that maps a string (the *name* of the dataset) to a tuple containing both the URL to locate the dataset and the SHA-1 key that verifies the integrity of the file. All such datasets are hosted at the site whose address is `DATA_URL`.

```
import hashlib
import os
import tarfile
import zipfile
import requests

#@save
DATA_HUB = dict()
DATA_URL = 'http://d2l-data.s3-accelerate.amazonaws.com/'
```

[73] https://discuss.d2l.ai/t/105
[74] https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names

The following `download` function downloads a dataset, caches it in a local directory (`../data` by default), and returns the name of the downloaded file. If a file corresponding to this dataset already exists in the cache directory and its SHA-1 matches the one stored in `DATA_HUB`, our code will use the cached file to avoid clogging up your internet with redundant downloads.

```
def download(name, cache_dir=os.path.join('..', 'data')):  #@save
    """Download a file inserted into DATA_HUB, return the local filename."""
    assert name in DATA_HUB, f"{name} does not exist in {DATA_HUB}."
    url, sha1_hash = DATA_HUB[name]
    os.makedirs(cache_dir, exist_ok=True)
    fname = os.path.join(cache_dir, url.split('/')[-1])
    if os.path.exists(fname):
        sha1 = hashlib.sha1()
        with open(fname, 'rb') as f:
            while True:
                data = f.read(1048576)
                if not data:
                    break
                sha1.update(data)
        if sha1.hexdigest() == sha1_hash:
            return fname  # Hit cache
    print(f'Downloading {fname} from {url}...')
    r = requests.get(url, stream=True, verify=True)
    with open(fname, 'wb') as f:
        f.write(r.content)
    return fname
```

We also implement two additional utility functions: one is to download and extract a zip or tar file and the other to download all the datasets used in this book from `DATA_HUB` into the cache directory.

```
def download_extract(name, folder=None):  #@save
    """Download and extract a zip/tar file."""
    fname = download(name)
    base_dir = os.path.dirname(fname)
    data_dir, ext = os.path.splitext(fname)
    if ext == '.zip':
        fp = zipfile.ZipFile(fname, 'r')
    elif ext in ('.tar', '.gz'):
        fp = tarfile.open(fname, 'r')
    else:
        assert False, 'Only zip/tar files can be extracted.'
    fp.extractall(base_dir)
    return os.path.join(base_dir, folder) if folder else data_dir

def download_all():  #@save
    """Download all files in the DATA_HUB."""
    for name in DATA_HUB:
        download(name)
```

4.10.2 Kaggle

Kaggle[75] is a popular platform that hosts machine learning competitions. Each competition centers on a dataset and many are sponsored by stakeholders who offer prizes to the winning solutions. The platform helps users to interact via forums and shared code, fostering both collaboration and competition. While leaderboard chasing often spirals out of control, with researchers focusing myopically on preprocessing steps rather than asking fundamental questions, there is also tremendous value in the objectivity of a platform that facilitates direct quantitative comparisons among competing approaches as well as code sharing so that everyone can learn what did and did not work. If you want to participate in a Kaggle competition, you will first need to register for an account (see Fig. 4.10.1).

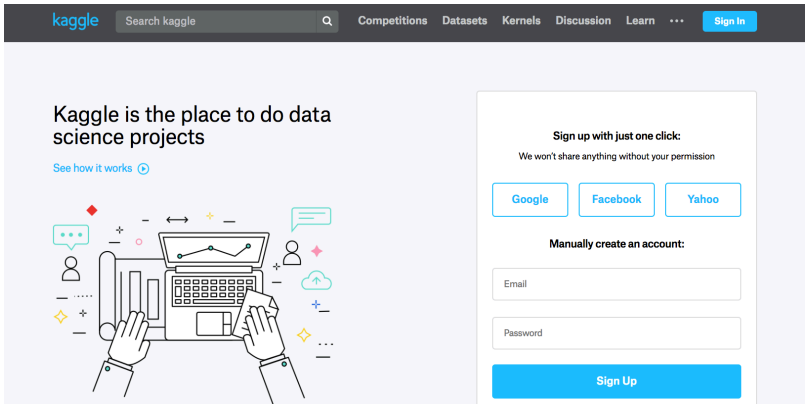


Fig. 4.10.1: The Kaggle website.

On the house price prediction competition page, as illustrated in Fig. 4.10.2, you can find the dataset (under the "Data" tab), submit predictions, and see your ranking, The URL is right here:

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

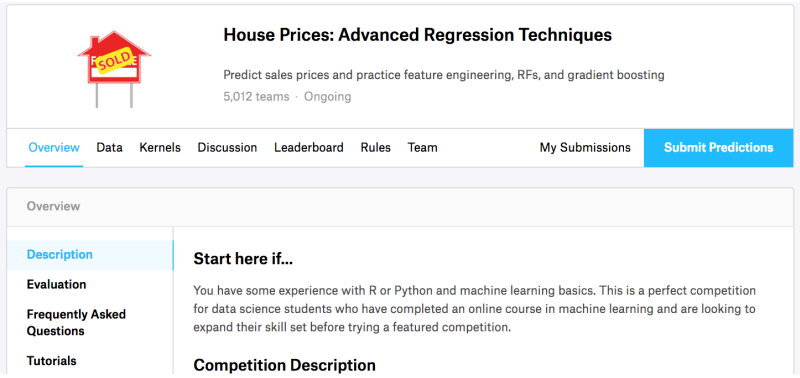


Fig. 4.10.2: The house price prediction competition page.

[75] https://www.kaggle.com

4.10.3 Accessing and Reading the Dataset

Note that the competition data is separated into training and test sets. Each record includes the property value of the house and attributes such as street type, year of construction, roof type, basement condition, etc. The features consist of various data types. For example, the year of construction is represented by an integer, the roof type by discrete categorical assignments, and other features by floating point numbers. And here is where reality complicates things: for some examples, some data are altogether missing with the missing value marked simply as "na". The price of each house is included for the training set only (it is a competition after all). We will want to partition the training set to create a validation set, but we only get to evaluate our models on the official test set after uploading predictions to Kaggle. The "Data" tab on the competition tab in Fig. 4.10.2 has links to download the data.

To get started, we will read in and process the data using pandas, which we have introduced in Section 2.2. So, you will want to make sure that you have pandas installed before proceeding further. Fortunately, if you are reading in Jupyter, we can install pandas without even leaving the notebook.

```
# If pandas is not installed, please uncomment the following line:
# !pip install pandas

%matplotlib inline
import pandas as pd
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

For convenience, we can download and cache the Kaggle housing dataset using the script we defined above.

```
DATA_HUB['kaggle_house_train'] = (  #@save
    DATA_URL + 'kaggle_house_pred_train.csv',
    '585e9cc93e70b39160e7921475f9bcd7d31219ce')

DATA_HUB['kaggle_house_test'] = (  #@save
    DATA_URL + 'kaggle_house_pred_test.csv',
    'fa19780a7b011d9b009e8bff8e99922a8ee2eb90')
```

We use pandas to load the two csv files containing training and test data respectively.

```
train_data = pd.read_csv(download('kaggle_house_train'))
test_data = pd.read_csv(download('kaggle_house_test'))
```

```
Downloading ../data/kaggle_house_pred_train.csv from http://d2l-data.s3-accelerate.amazonaws.
↪com/kaggle_house_pred_train.csv...
Downloading ../data/kaggle_house_pred_test.csv from http://d2l-data.s3-accelerate.amazonaws.
↪com/kaggle_house_pred_test.csv...
```

The training dataset includes 1460 examples, 80 features, and 1 label, while the test data contains 1459 examples and 80 features.

```
print(train_data.shape)
print(test_data.shape)
```

```
(1460, 81)
(1459, 80)
```

Let us take a look at the first four and last two features as well as the label (SalePrice) from the first four examples.

```
print(train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
0   1          60       RL         65.0       WD        Normal     208500
1   2          20       RL         80.0       WD        Normal     181500
2   3          60       RL         68.0       WD        Normal     223500
3   4          70       RL         60.0       WD       Abnorml     140000
```

We can see that in each example, the first feature is the ID. This helps the model identify each training example. While this is convenient, it does not carry any information for prediction purposes. Hence, we remove it from the dataset before feeding the data into the model.

```
all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

4.10.4 Data Preprocessing

As stated above, we have a wide variety of data types. We will need to preprocess the data before we can start modeling. Let us start with the numerical features. First, we apply a heuristic, replacing all missing values by the corresponding feature's mean. Then, to put all features on a common scale, we *standardize* the data by rescaling features to zero mean and unit variance:

$$x \leftarrow \frac{x - \mu}{\sigma}, \tag{4.10.1}$$

where μ and σ denote mean and standard deviation, respectively. To verify that this indeed transforms our feature (variable) such that it has zero mean and unit variance, note that $E[\frac{x-\mu}{\sigma}] = \frac{\mu-\mu}{\sigma} = 0$ and that $E[(x-\mu)^2] = (\sigma^2 + \mu^2) - 2\mu^2 + \mu^2 = \sigma^2$. Intuitively, we standardize the data for two reasons. First, it proves convenient for optimization. Second, because we do not know *a priori* which features will be relevant, we do not want to penalize coefficients assigned to one feature more than on any other.

```
# If test data were inaccessible, mean and standard deviation could be
# calculated from training data
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# After standardizing the data all means vanish, hence we can set missing
# values to 0
all_features[numeric_features] = all_features[numeric_features].fillna(0)
```

Next we deal with discrete values. This includes features such as "MSZoning". We replace them by a one-hot encoding in the same way that we previously transformed multiclass labels into vectors

(see Section 3.4.1). For instance, "MSZoning" assumes the values "RL" and "RM". Dropping the "MSZoning" feature, two new indicator features "MSZoning_RL" and "MSZoning_RM" are created with values being either 0 or 1. According to one-hot encoding, if the original value of "MSZoning" is "RL", then "MSZoning_RL" is 1 and "MSZoning_RM" is 0. The pandas package does this automatically for us.

```
# `Dummy_na=True` considers "na" (missing value) as a valid feature value, and
# creates an indicator feature for it
all_features = pd.get_dummies(all_features, dummy_na=True)
all_features.shape
```

```
(2919, 331)
```

You can see that this conversion increases the number of features from 79 to 331. Finally, via the values attribute, we can extract the NumPy format from the pandas format and convert it into the tensor representation for training.

```
n_train = train_data.shape[0]
train_features = np.array(all_features[:n_train].values, dtype=np.float32)
test_features = np.array(all_features[n_train:].values, dtype=np.float32)
train_labels = np.array(train_data.SalePrice.values.reshape(-1, 1),
                        dtype=np.float32)
```

4.10.5 Training

To get started we train a linear model with squared loss. Not surprisingly, our linear model will not lead to a competition-winning submission but it provides a sanity check to see whether there is meaningful information in the data. If we cannot do better than random guessing here, then there might be a good chance that we have a data processing bug. And if things work, the linear model will serve as a baseline giving us some intuition about how close the simple model gets to the best reported models, giving us a sense of how much gain we should expect from fancier models.

```
loss = gluon.loss.L2Loss()

def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize()
    return net
```

With house prices, as with stock prices, we care about relative quantities more than absolute quantities. Thus we tend to care more about the relative error $\frac{y-\hat{y}}{y}$ than about the absolute error $y - \hat{y}$. For instance, if our prediction is off by USD 100,000 when estimating the price of a house in Rural Ohio, where the value of a typical house is 125,000 USD, then we are probably doing a horrible job. On the other hand, if we err by this amount in Los Altos Hills, California, this might represent a stunningly accurate prediction (there, the median house price exceeds 4 million USD).

One way to address this problem is to measure the discrepancy in the logarithm of the price estimates. In fact, this is also the official error measure used by the competition to evaluate the quality of submissions. After all, a small value δ for $|\log y - \log \hat{y}| \leq \delta$ translates into $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$. This

leads to the following root-mean-squared-error between the logarithm of the predicted price and the logarithm of the label price:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log y_i - \log \hat{y}_i)^2}. \tag{4.10.2}$$

```
def log_rmse(net, features, labels):
    # To further stabilize the value when the logarithm is taken, set the
    # value less than 1 as 1
    clipped_preds = np.clip(net(features), 1, float('inf'))
    return np.sqrt(2 * loss(np.log(clipped_preds), np.log(labels)).mean())
```

Unlike in previous sections, our training functions will rely on the Adam optimizer (we will describe it in greater detail later). The main appeal of this optimizer is that, despite doing no better (and sometimes worse) given unlimited resources for hyperparameter optimization, people tend to find that it is significantly less sensitive to the initial learning rate.

```
def train(net, train_features, train_labels, test_features, test_labels,
          num_epochs, learning_rate, weight_decay, batch_size):
    train_ls, test_ls = [], []
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    # The Adam optimization algorithm is used here
    trainer = gluon.Trainer(net.collect_params(), 'adam', {
        'learning_rate': learning_rate,
        'wd': weight_decay})
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        train_ls.append(log_rmse(net, train_features, train_labels))
        if test_labels is not None:
            test_ls.append(log_rmse(net, test_features, test_labels))
    return train_ls, test_ls
```

4.10.6 K-Fold Cross-Validation

You might recall that we introduced K-fold cross-validation in the section where we discussed how to deal with model selection (Section 4.4). We will put this to good use to select the model design and to adjust the hyperparameters. We first need a function that returns the i^{th} fold of the data in a K-fold cross-validation procedure. It proceeds by slicing out the i^{th} segment as validation data and returning the rest as training data. Note that this is not the most efficient way of handling data and we would definitely do something much smarter if our dataset was considerably larger. But this added complexity might obfuscate our code unnecessarily so we can safely omit it here owing to the simplicity of our problem.

```
def get_k_fold_data(k, i, X, y):
    assert k > 1
    fold_size = X.shape[0] // k
    X_train, y_train = None, None
```

(continues on next page)

(continued from previous page)

```
    for j in range(k):
        idx = slice(j * fold_size, (j + 1) * fold_size)
        X_part, y_part = X[idx, :], y[idx]
        if j == i:
            X_valid, y_valid = X_part, y_part
        elif X_train is None:
            X_train, y_train = X_part, y_part
        else:
            X_train = np.concatenate([X_train, X_part], 0)
            y_train = np.concatenate([y_train, y_part], 0)
    return X_train, y_train, X_valid, y_valid
```

The training and verification error averages are returned when we train K times in the K-fold cross-validation.

```
def k_fold(k, X_train, y_train, num_epochs, learning_rate, weight_decay,
           batch_size):
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
        data = get_k_fold_data(k, i, X_train, y_train)
        net = get_net()
        train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
                                   weight_decay, batch_size)
        train_l_sum += train_ls[-1]
        valid_l_sum += valid_ls[-1]
        if i == 0:
            d2l.plot(list(range(1, num_epochs + 1)), [train_ls, valid_ls],
                     xlabel='epoch', ylabel='rmse', xlim=[1, num_epochs],
                     legend=['train', 'valid'], yscale='log')
        print(f'fold {i + 1}, train log rmse {float(train_ls[-1]):f}, '
              f'valid log rmse {float(valid_ls[-1]):f}')
    return train_l_sum / k, valid_l_sum / k
```

4.10.7 Model Selection

In this example, we pick an untuned set of hyperparameters and leave it up to the reader to improve the model. Finding a good choice can take time, depending on how many variables one optimizes over. With a large enough dataset, and the normal sorts of hyperparameters, K-fold cross-validation tends to be reasonably resilient against multiple testing. However, if we try an unreasonably large number of options we might just get lucky and find that our validation performance is no longer representative of the true error.

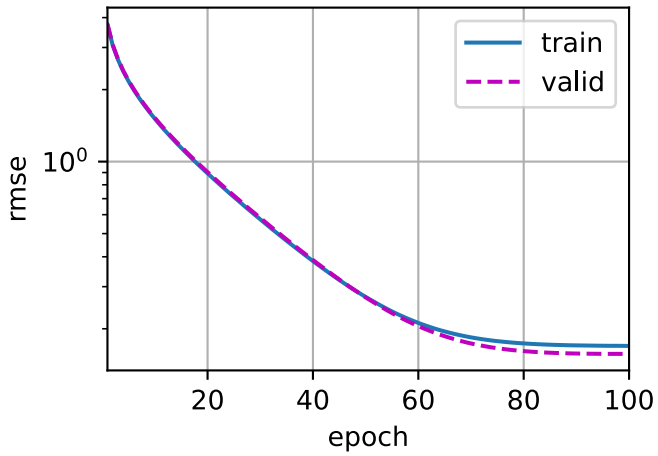
```
k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64
train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                          weight_decay, batch_size)
print(f'{k}-fold validation: avg train log rmse: {float(train_l):f}, '
      f'avg valid log rmse: {float(valid_l):f}')
```

```
fold 1, train log rmse 0.169552, valid log rmse 0.156981
fold 2, train log rmse 0.162441, valid log rmse 0.190252
fold 3, train log rmse 0.163629, valid log rmse 0.168082
```

(continues on next page)

(continued from previous page)

```
fold 4, train log rmse 0.167246, valid log rmse 0.154274
fold 5, train log rmse 0.162747, valid log rmse 0.182818
5-fold validation: avg train log rmse: 0.165123, avg valid log rmse: 0.170481
```



Notice that sometimes the number of training errors for a set of hyperparameters can be very low, even as the number of errors on K-fold cross-validation is considerably higher. This indicates that we are overfitting. Throughout training you will want to monitor both numbers. Less overfitting might indicate that our data can support a more powerful model. Massive overfitting might suggest that we can gain by incorporating regularization techniques.

4.10.8 Submitting Predictions on Kaggle

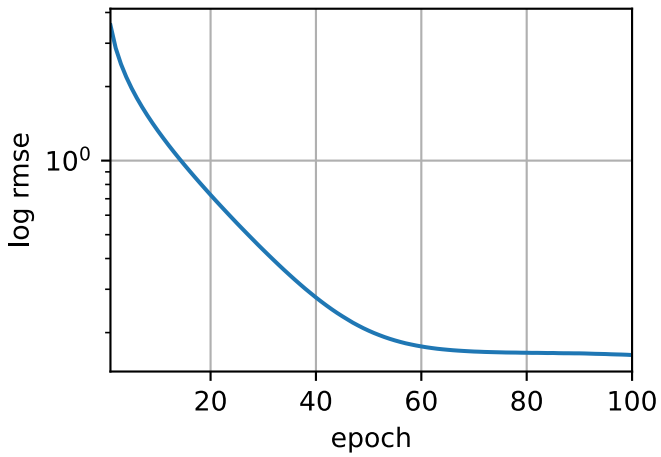
Now that we know what a good choice of hyperparameters should be, we might as well use all the data to train on it (rather than just $1 - 1/K$ of the data that are used in the cross-validation slices). The model that we obtain in this way can then be applied to the test set. Saving the predictions in a csv file will simplify uploading the results to Kaggle.

```
def train_and_pred(train_features, test_feature, train_labels, test_data,
                   num_epochs, lr, weight_decay, batch_size):
    net = get_net()
    train_ls, _ = train(net, train_features, train_labels, None, None,
                        num_epochs, lr, weight_decay, batch_size)
    d2l.plot(np.arange(1, num_epochs + 1), [train_ls], xlabel='epoch',
             ylabel='log rmse', xlim=[1, num_epochs], yscale='log')
    print(f'train log rmse {float(train_ls[-1]):f}')
    # Apply the network to the test set
    preds = net(test_features).asnumpy()
    # Reformat it to export to Kaggle
    test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
    submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
    submission.to_csv('submission.csv', index=False)
```

One nice sanity check is to see whether the predictions on the test set resemble those of the K-fold cross-validation process. If they do, it is time to upload them to Kaggle. The following code will generate a file called `submission.csv`.

```
train_and_pred(train_features, test_features, train_labels, test_data,
               num_epochs, lr, weight_decay, batch_size)
```

```
train log rmse 0.162116
```



Next, as demonstrated in Fig. 4.10.3, we can submit our predictions on Kaggle and see how they compare with the actual house prices (labels) on the test set. The steps are quite simple:

- Log in to the Kaggle website and visit the house price prediction competition page.
- Click the "Submit Predictions" or "Late Submission" button (as of this writing, the button is located on the right).
- Click the "Upload Submission File" button in the dashed box at the bottom of the page and select the prediction file you wish to upload.
- Click the "Make Submission" button at the bottom of the page to view your results.

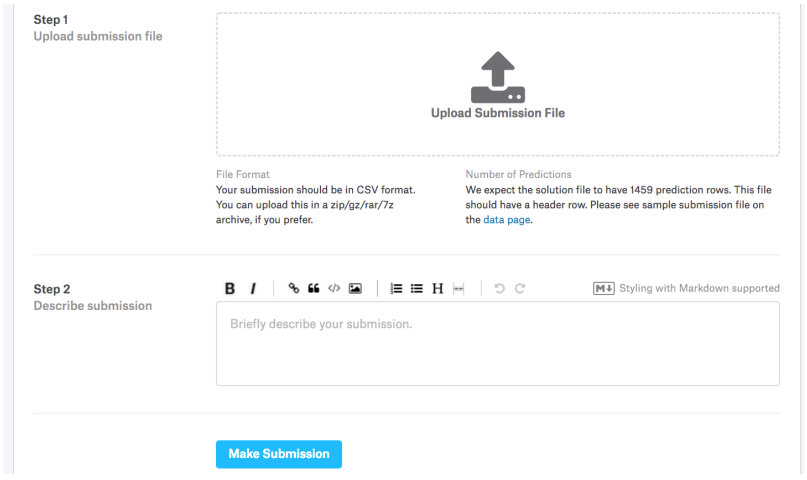


Fig. 4.10.3: Submitting data to Kaggle

Summary

- Real data often contain a mix of different data types and need to be preprocessed.
- Rescaling real-valued data to zero mean and unit variance is a good default. So is replacing missing values with their mean.
- Transforming categorical features into indicator features allows us to treat them like one-hot vectors.
- We can use K-fold cross-validation to select the model and adjust the hyperparameters.
- Logarithms are useful for relative errors.

Exercises

1. Submit your predictions for this section to Kaggle. How good are your predictions?
2. Can you improve your model by minimizing the logarithm of prices directly? What happens if you try to predict the logarithm of the price rather than the price?
3. Is it always a good idea to replace missing values by their mean? Hint: can you construct a situation where the values are not missing at random?
4. Improve the score on Kaggle by tuning the hyperparameters through K-fold cross-validation.
5. Improve the score by improving the model (e.g., layers, weight decay, and dropout).
6. What happens if we do not standardize the continuous numerical features like what we have done in this section?

Discussions[76]

[76] https://discuss.d2l.ai/t/106

5 | Deep Learning Computation

Alongside giant datasets and powerful hardware, great software tools have played an indispensable role in the rapid progress of deep learning. Starting with the pathbreaking Theano library released in 2007, flexible open-source tools have enabled researchers to rapidly prototype models, avoiding repetitive work when recycling standard components while still maintaining the ability to make low-level modifications. Over time, deep learning's libraries have evolved to offer increasingly coarse abstractions. Just as semiconductor designers went from specifying transistors to logical circuits to writing code, neural networks researchers have moved from thinking about the behavior of individual artificial neurons to conceiving of networks in terms of whole layers, and now often design architectures with far coarser *blocks* in mind.

So far, we have introduced some basic machine learning concepts, ramping up to fully-functional deep learning models. In the last chapter, we implemented each component of an MLP from scratch and even showed how to leverage high-level APIs to roll out the same models effortlessly. To get you that far that fast, we *called upon* the libraries, but skipped over more advanced details about *how they work*. In this chapter, we will peel back the curtain, digging deeper into the key components of deep learning computation, namely model construction, parameter access and initialization, designing custom layers and blocks, reading and writing models to disk, and leveraging GPUs to achieve dramatic speedups. These insights will move you from *end user* to *power user*, giving you the tools needed to reap the benefits of a mature deep learning library while retaining the flexibility to implement more complex models, including those you invent yourself! While this chapter does not introduce any new models or datasets, the advanced modeling chapters that follow rely heavily on these techniques.

5.1 Layers and Blocks

When we first introduced neural networks, we focused on linear models with a single output. Here, the entire model consists of just a single neuron. Note that a single neuron (i) takes some set of inputs; (ii) generates a corresponding scalar output; and (iii) has a set of associated parameters that can be updated to optimize some objective function of interest. Then, once we started thinking about networks with multiple outputs, we leveraged vectorized arithmetic to characterize an entire layer of neurons. Just like individual neurons, layers (i) take a set of inputs, (ii) generate corresponding outputs, and (iii) are described by a set of tunable parameters. When we worked through softmax regression, a single layer was itself the model. However, even when we subsequently introduced MLPs, we could still think of the model as retaining this same basic structure.

Interestingly, for MLPs, both the entire model and its constituent layers share this structure. The entire model takes in raw inputs (the features), generates outputs (the predictions), and possesses parameters (the combined parameters from all constituent layers). Likewise, each individual layer ingests inputs (supplied by the previous layer) generates outputs (the inputs to the subsequent

layer), and possesses a set of tunable parameters that are updated according to the signal that flows backwards from the subsequent layer.

While you might think that neurons, layers, and models give us enough abstractions to go about our business, it turns out that we often find it convenient to speak about components that are larger than an individual layer but smaller than the entire model. For example, the ResNet-152 architecture, which is wildly popular in computer vision, possesses hundreds of layers. These layers consist of repeating patterns of *groups of layers*. Implementing such a network one layer at a time can grow tedious. This concern is not just hypothetical—such design patterns are common in practice. The ResNet architecture mentioned above won the 2015 ImageNet and COCO computer vision competitions for both recognition and detection (He et al., 2016a) and remains a go-to architecture for many vision tasks. Similar architectures in which layers are arranged in various repeating patterns are now ubiquitous in other domains, including natural language processing and speech.

To implement these complex networks, we introduce the concept of a neural network *block*. A block could describe a single layer, a component consisting of multiple layers, or the entire model itself! One benefit of working with the block abstraction is that they can be combined into larger artifacts, often recursively. This is illustrated in Fig. 5.1.1. By defining code to generate blocks of arbitrary complexity on demand, we can write surprisingly compact code and still implement complex neural networks.

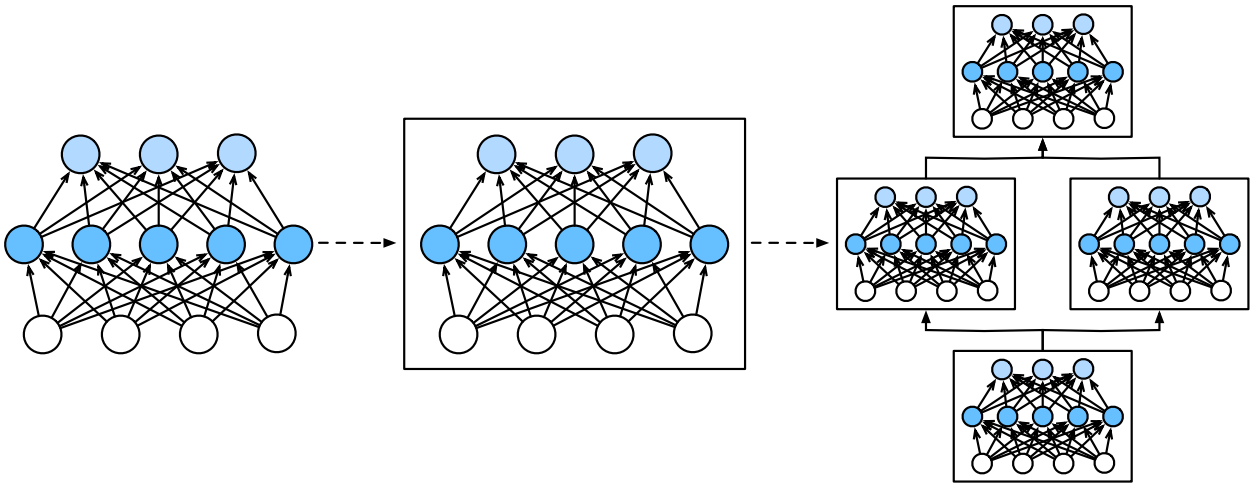


Fig. 5.1.1: Multiple layers are combined into blocks, forming repeating patterns of larger models.

From a programing standpoint, a block is represented by a *class*. Any subclass of it must define a forward propagation function that transforms its input into output and must store any necessary parameters. Note that some blocks do not require any parameters at all. Finally a block must possess a backpropagation function, for purposes of calculating gradients. Fortunately, due to some behind-the-scenes magic supplied by the auto differentiation (introduced in Section 2.5) when defining our own block, we only need to worry about parameters and the forward propagation function.

To begin, we revisit the code that we used to implement MLPs (Section 4.3). The following code generates a network with one fully-connected hidden layer with 256 units and ReLU activation, followed by a fully-connected output layer with 10 units (no activation function).

```
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()

net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()

X = np.random.uniform(size=(2, 20))
net(X)
```

```
array([[ 0.06240274, -0.03268593,  0.02582653,  0.02254181, -0.03728798,
        -0.04253785,  0.00540612, -0.01364185, -0.09915454, -0.02272737],
       [ 0.02816679, -0.03341204,  0.03565665,  0.02506384, -0.04136416,
        -0.04941844,  0.01738529,  0.01081963, -0.09932579, -0.01176296]])
```

In this example, we constructed our model by instantiating an `nn.Sequential`, assigning the returned object to the `net` variable. Next, we repeatedly call its `add` function, appending layers in the order that they should be executed. In short, `nn.Sequential` defines a special kind of `Block`, the class that presents a block in Gluon. It maintains an ordered list of constituent `Blocks`. The `add` function simply facilitates the addition of each successive `Block` to the list. Note that each layer is an instance of the `Dense` class which is itself a subclass of `Block`. The forward propagation (`forward`) function is also remarkably simple: it chains each `Block` in the list together, passing the output of each as the input to the next. Note that until now, we have been invoking our models via the construction `net(X)` to obtain their outputs. This is actually just shorthand for `net.forward(X)`, a slick Python trick achieved via the `Block` class's `__call__` function.

5.1.1 A Custom Block

Perhaps the easiest way to develop intuition about how a block works is to implement one ourselves. Before we implement our own custom block, we briefly summarize the basic functionality that each block must provide:

1. Ingest input data as arguments to its forward propagation function.
2. Generate an output by having the forward propagation function return a value. Note that the output may have a different shape from the input. For example, the first fully-connected layer in our model above ingests an input of arbitrary dimension but returns an output of dimension 256.
3. Calculate the gradient of its output with respect to its input, which can be accessed via its backpropagation function. Typically this happens automatically.
4. Store and provide access to those parameters necessary to execute the forward propagation computation.
5. Initialize model parameters as needed.

In the following snippet, we code up a block from scratch corresponding to an MLP with one hidden layer with 256 hidden units, and a 10-dimensional output layer. Note that the `MLP` class below inherits the class that represents a block. We will heavily rely on the parent class's functions, sup-

plying only our own constructor (the `__init__` function in Python) and the forward propagation function.

```
class MLP(nn.Block):
    # Declare a layer with model parameters. Here, we declare two
    # fully-connected layers
    def __init__(self, **kwargs):
        # Call the constructor of the `MLP` parent class `Block` to perform
        # the necessary initialization. In this way, other function arguments
        # can also be specified during class instantiation, such as the model
        # parameters, `params` (to be described later)
        super().__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu')  # Hidden layer
        self.out = nn.Dense(10)  # Output layer

    # Define the forward propagation of the model, that is, how to return the
    # required model output based on the input `X`
    def forward(self, X):
        return self.out(self.hidden(X))
```

Let us first focus on the forward propagation function. Note that it takes `X` as the input, calculates the hidden representation with the activation function applied, and outputs its logits. In this `MLP` implementation, both layers are instance variables. To see why this is reasonable, imagine instantiating two MLPs, `net1` and `net2`, and training them on different data. Naturally, we would expect them to represent two different learned models.

We instantiate the MLP's layers in the constructor and subsequently invoke these layers on each call to the forward propagation function. Note a few key details. First, our customized `__init__` function invokes the parent class's `__init__` function via `super().__init__()` sparing us the pain of restating boilerplate code applicable to most blocks. We then instantiate our two fully-connected layers, assigning them to `self.hidden` and `self.out`. Note that unless we implement a new operator, we need not worry about the backpropagation function or parameter initialization. The system will generate these functions automatically. Let us try this out.

```
net = MLP()
net.initialize()
net(X)
```

```
array([[-0.03989595, -0.10414709,  0.06799038,  0.05245074,  0.0252606 ,
        -0.00640342,  0.04182098, -0.01665318, -0.02067345, -0.07863816],
       [-0.03612847, -0.07210435,  0.09159479,  0.07890773,  0.02494171,
        -0.01028665,  0.01732427, -0.02843244,  0.03772651, -0.06671703]])
```

A key virtue of the block abstraction is its versatility. We can subclass a block to create layers (such as the fully-connected layer class), entire models (such as the `MLP` class above), or various components of intermediate complexity. We exploit this versatility throughout the following chapters, such as when addressing convolutional neural networks.

5.1.2 The Sequential Block

We can now take a closer look at how the `Sequential` class works. Recall that `Sequential` was designed to daisy-chain other blocks together. To build our own simplified `MySequential`, we just need to define two key function: 1. A function to append blocks one by one to a list. 2. A forward propagation function to pass an input through the chain of blocks, in the same order as they were appended.

The following `MySequential` class delivers the same functionality of the default `Sequential` class.

```
class MySequential(nn.Block):
    def add(self, block):
        # Here, `block` is an instance of a `Block` subclass, and we assume
        # that it has a unique name. We save it in the member variable
        # `_children` of the `Block` class, and its type is OrderedDict. When
        # the `MySequential` instance calls the `initialize` function, the
        # system automatically initializes all members of `_children`
        self._children[block.name] = block

    def forward(self, X):
        # OrderedDict guarantees that members will be traversed in the order
        # they were added
        for block in self._children.values():
            X = block(X)
        return X
```

The `add` function adds a single block to the ordered dictionary `_children`. You might wonder why every Gluon `Block` possesses a `_children` attribute and why we used it rather than just define a Python list ourselves. In short the chief advantage of `_children` is that during our block's parameter initialization, Gluon knows to look inside the `_children` dictionary to find sub-blocks whose parameters also need to be initialized.

When our `MySequential`'s forward propagation function is invoked, each added block is executed in the order in which they were added. We can now reimplement an MLP using our `MySequential` class.

```
net = MySequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(X)
```

```
array([[-0.0764568 , -0.01130233,  0.04952145, -0.04651389, -0.04131571,
        -0.05884131, -0.06213811,  0.01311471, -0.01379425, -0.02514282],
       [-0.05124623,  0.00711232, -0.00155933, -0.07555379, -0.06675334,
        -0.01762914,  0.00589085,  0.0144719 , -0.04330775,  0.03317727]])
```

Note that this use of `MySequential` is identical to the code we previously wrote for the `Sequential` class (as described in Section 4.3).

5.1.3 Executing Code in the Forward Propagation Function

The `Sequential` class makes model construction easy, allowing us to assemble new architectures without having to define our own class. However, not all architectures are simple daisy chains. When greater flexibility is required, we will want to define our own blocks. For example, we might want to execute Python's control flow within the forward propagation function. Moreover, we might want to perform arbitrary mathematical operations, not simply relying on predefined neural network layers.

You might have noticed that until now, all of the operations in our networks have acted upon our network's activations and its parameters. Sometimes, however, we might want to incorporate terms that are neither the result of previous layers nor updatable parameters. We call these *constant parameters*. Say for example that we want a layer that calculates the function $f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}$, where $\mathbf{x}$ is the input, $\mathbf{w}$ is our parameter, and c is some specified constant that is not updated during optimization. So we implement a `FixedHiddenMLP` class as follows.

```
class FixedHiddenMLP(nn.Block):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # Random weight parameters created with the `get_constant` function
        # are not updated during training (i.e., constant parameters)
        self.rand_weight = self.params.get_constant(
            'rand_weight', np.random.uniform(size=(20, 20)))
        self.dense = nn.Dense(20, activation='relu')

    def forward(self, X):
        X = self.dense(X)
        # Use the created constant parameters, as well as the `relu` and `dot`
        # functions
        X = npx.relu(np.dot(X, self.rand_weight.data()) + 1)
        # Reuse the fully-connected layer. This is equivalent to sharing
        # parameters with two fully-connected layers
        X = self.dense(X)
        # Control flow
        while np.abs(X).sum() > 1:
            X /= 2
        return X.sum()
```

In this `FixedHiddenMLP` model, we implement a hidden layer whose weights (`self.rand_weight`) are initialized randomly at instantiation and are thereafter constant. This weight is not a model parameter and thus it is never updated by backpropagation. The network then passes the output of this "fixed" layer through a fully-connected layer.

Note that before returning the output, our model did something unusual. We ran a while-loop, testing on the condition its L_1 norm is larger than 1, and dividing our output vector by 2 until it satisfied the condition. Finally, we returned the sum of the entries in `X`. To our knowledge, no standard neural network performs this operation. Note that this particular operation may not be useful in any real-world task. Our point is only to show you how to integrate arbitrary code into the flow of your neural network computations.

```
net = FixedHiddenMLP()
net.initialize()
net(X)
```

```
array(0.52637565)
```

We can mix and match various ways of assembling blocks together. In the following example, we nest blocks in some creative ways.

```
class NestMLP(nn.Block):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.net = nn.Sequential()
        self.net.add(nn.Dense(64, activation='relu'),
                     nn.Dense(32, activation='relu'))
        self.dense = nn.Dense(16, activation='relu')

    def forward(self, X):
        return self.dense(self.net(X))

chimera = nn.Sequential()
chimera.add(NestMLP(), nn.Dense(20), FixedHiddenMLP())
chimera.initialize()
chimera(X)
```

```
array(0.9772054)
```

5.1.4 Efficiency

The avid reader might start to worry about the efficiency of some of these operations. After all, we have lots of dictionary lookups, code execution, and lots of other Pythonic things taking place in what is supposed to be a high-performance deep learning library. The problems of Python's global interpreter lock[77] are well known. In the context of deep learning, we may worry that our extremely fast GPU(s) might have to wait until a puny CPU runs Python code before it gets another job to run. The best way to speed up Python is by avoiding it altogether.

One way that Gluon does this is by allowing for *hybridization*, which will be described later. Here, the Python interpreter executes a block the first time it is invoked. The Gluon runtime records what is happening and the next time around it short-circuits calls to Python. This can accelerate things considerably in some cases but care needs to be taken when control flow (as above) leads down different branches on different passes through the net. We recommend that the interested reader checks out the hybridization section (Section 12.1) to learn about compilation after finishing the current chapter.

[77] https://wiki.python.org/moin/GlobalInterpreterLock

Summary

- Layers are blocks.
- Many layers can comprise a block.
- Many blocks can comprise a block.
- A block can contain code.
- Blocks take care of lots of housekeeping, including parameter initialization and backpropagation.
- Sequential concatenations of layers and blocks are handled by the `Sequential` block.

Exercises

1. What kinds of problems will occur if you change `MySequential` to store blocks in a Python list?
2. Implement a block that takes two blocks as an argument, say `net1` and `net2` and returns the concatenated output of both networks in the forward propagation. This is also called a parallel block.
3. Assume that you want to concatenate multiple instances of the same network. Implement a factory function that generates multiple instances of the same block and build a larger network from it.

Discussions[78]

5.2 Parameter Management

Once we have chosen an architecture and set our hyperparameters, we proceed to the training loop, where our goal is to find parameter values that minimize our loss function. After training, we will need these parameters in order to make future predictions. Additionally, we will sometimes wish to extract the parameters either to reuse them in some other context, to save our model to disk so that it may be executed in other software, or for examination in the hope of gaining scientific understanding.

Most of the time, we will be able to ignore the nitty-gritty details of how parameters are declared and manipulated, relying on deep learning frameworks to do the heavy lifting. However, when we move away from stacked architectures with standard layers, we will sometimes need to get into the weeds of declaring and manipulating parameters. In this section, we cover the following:

- Accessing parameters for debugging, diagnostics, and visualizations.
- Parameter initialization.
- Sharing parameters across different model components.

We start by focusing on an MLP with one hidden layer.

[78] https://discuss.d2l.ai/t/54

```
from mxnet import init, np, npx
from mxnet.gluon import nn

npx.set_np()

net = nn.Sequential()
net.add(nn.Dense(8, activation='relu'))
net.add(nn.Dense(1))
net.initialize()  # Use the default initialization method

X = np.random.uniform(size=(2, 4))
net(X)  # Forward computation
```

```
array([[0.0054572 ],
       [0.00488594]])
```

5.2.1 Parameter Access

Let us start with how to access parameters from the models that you already know. When a model is defined via the `Sequential` class, we can first access any layer by indexing into the model as though it were a list. Each layer's parameters are conveniently located in its attribute. We can inspect the parameters of the second fully-connected layer as follows.

```
print(net[1].params)
```

```
dense1_ (
  Parameter dense1_weight (shape=(1, 8), dtype=float32)
  Parameter dense1_bias (shape=(1,), dtype=float32)
)
```

The output tells us a few important things. First, this fully-connected layer contains two parameters, corresponding to that layer's weights and biases, respectively. Both are stored as single precision floats (float32). Note that the names of the parameters allow us to uniquely identify each layer's parameters, even in a network containing hundreds of layers.

Targeted Parameters

Note that each parameter is represented as an instance of the parameter class. To do anything useful with the parameters, we first need to access the underlying numerical values. There are several ways to do this. Some are simpler while others are more general. The following code extracts the bias from the second neural network layer, which returns a parameter class instance, and further accesses that parameter's value.

```
print(type(net[1].bias))
print(net[1].bias)
print(net[1].bias.data())
```

```
<class 'mxnet.gluon.parameter.Parameter'>
Parameter dense1_bias (shape=(1,), dtype=float32)
[0.]
```

Parameters are complex objects, containing values, gradients, and additional information. That's why we need to request the value explicitly.

In addition to the value, each parameter also allows us to access the gradient. Because we have not invoked backpropagation for this network yet, it is in its initial state.

```
net[1].weight.grad()
```

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

All Parameters at Once

When we need to perform operations on all parameters, accessing them one-by-one can grow tedious. The situation can grow especially unwieldy when we work with more complex blocks (e.g., nested blocks), since we would need to recurse through the entire tree to extract each sub-block's parameters. Below we demonstrate accessing the parameters of the first fully-connected layer vs. accessing all layers.

```
print(net[0].collect_params())
print(net.collect_params())
```

```
dense0_ (
  Parameter dense0_weight (shape=(8, 4), dtype=float32)
  Parameter dense0_bias (shape=(8,), dtype=float32)
)
sequential0_ (
  Parameter dense0_weight (shape=(8, 4), dtype=float32)
  Parameter dense0_bias (shape=(8,), dtype=float32)
  Parameter dense1_weight (shape=(1, 8), dtype=float32)
  Parameter dense1_bias (shape=(1,), dtype=float32)
)
```

This provides us with another way of accessing the parameters of the network as follows.

```
net.collect_params()['dense1_bias'].data()
```

```
array([0.])
```

Collecting Parameters from Nested Blocks

Let us see how the parameter naming conventions work if we nest multiple blocks inside each other. For that we first define a function that produces blocks (a block factory, so to speak) and then combine these inside yet larger blocks.

```
def block1():
    net = nn.Sequential()
    net.add(nn.Dense(32, activation='relu'))
    net.add(nn.Dense(16, activation='relu'))
    return net

def block2():
    net = nn.Sequential()
    for _ in range(4):
        # Nested here
        net.add(block1())
    return net

rgnet = nn.Sequential()
rgnet.add(block2())
rgnet.add(nn.Dense(10))
rgnet.initialize()
rgnet(X)
```

```
array([[-6.3465846e-09, -1.1096752e-09,  6.4161787e-09,  6.6354140e-09,
        -1.1265507e-09,  1.3284951e-10,  9.3619388e-09,  3.2229084e-09,
         5.9429879e-09,  8.8181435e-09],
       [-8.6219423e-09, -7.5150686e-10,  8.3133251e-09,  8.9321128e-09,
        -1.6740003e-09,  3.2405989e-10,  1.2115976e-08,  4.4926449e-09,
         8.0741742e-09,  1.2075874e-08]])
```

Now that we have designed the network, let us see how it is organized.

```
print(rgnet.collect_params)
print(rgnet.collect_params())
```

```
<bound method Block.collect_params of Sequential(
  (0): Sequential(
    (0): Sequential(
      (0): Dense(4 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (1): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (2): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (3): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
```

(continues on next page)

(continued from previous page)

```
    )
  )
  (1): Dense(16 -> 10, linear)
)>
sequential1_ (
  Parameter dense2_weight (shape=(32, 4), dtype=float32)
  Parameter dense2_bias (shape=(32,), dtype=float32)
  Parameter dense3_weight (shape=(16, 32), dtype=float32)
  Parameter dense3_bias (shape=(16,), dtype=float32)
  Parameter dense4_weight (shape=(32, 16), dtype=float32)
  Parameter dense4_bias (shape=(32,), dtype=float32)
  Parameter dense5_weight (shape=(16, 32), dtype=float32)
  Parameter dense5_bias (shape=(16,), dtype=float32)
  Parameter dense6_weight (shape=(32, 16), dtype=float32)
  Parameter dense6_bias (shape=(32,), dtype=float32)
  Parameter dense7_weight (shape=(16, 32), dtype=float32)
  Parameter dense7_bias (shape=(16,), dtype=float32)
  Parameter dense8_weight (shape=(32, 16), dtype=float32)
  Parameter dense8_bias (shape=(32,), dtype=float32)
  Parameter dense9_weight (shape=(16, 32), dtype=float32)
  Parameter dense9_bias (shape=(16,), dtype=float32)
  Parameter dense10_weight (shape=(10, 16), dtype=float32)
  Parameter dense10_bias (shape=(10,), dtype=float32)
)
```

Since the layers are hierarchically nested, we can also access them as though indexing through nested lists. For instance, we can access the first major block, within it the second sub-block, and within that the bias of the first layer, with as follows.

```
rgnet[0][1][0].bias.data()
```

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

5.2.2 Parameter Initialization

Now that we know how to access the parameters, let us look at how to initialize them properly. We discussed the need for proper initialization in Section 4.8. The deep learning framework provides default random initializations to its layers. However, we often want to initialize our weights according to various other protocols. The framework provides most commonly used protocols, and also allows to create a custom initializer.

By default, MXNet initializes weight parameters by randomly drawing from a uniform distribution $U(-0.07, 0.07)$, clearing bias parameters to zero. MXNet's `init` module provides a variety of preset initialization methods.

Built-in Initialization

Let us begin by calling on built-in initializers. The code below initializes all weight parameters as Gaussian random variables with standard deviation 0.01, while bias parameters cleared to zero.

```
# Here `force_reinit` ensures that parameters are freshly initialized even if
# they were already initialized previously
net.initialize(init=init.Normal(sigma=0.01), force_reinit=True)
net[0].weight.data()[0]
```

```
array([-0.00324057, -0.00895028, -0.00698632,  0.01030831])
```

We can also initialize all the parameters to a given constant value (say, 1).

```
net.initialize(init=init.Constant(1), force_reinit=True)
net[0].weight.data()[0]
```

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

We can also apply different initializers for certain blocks. For example, below we initialize the first layer with the Xavier initializer and initialize the second layer to a constant value of 42.

```
net[0].weight.initialize(init=init.Xavier(), force_reinit=True)
net[1].initialize(init=init.Constant(42), force_reinit=True)
print(net[0].weight.data()[0])
print(net[1].weight.data())
```

```
[-0.17594433  0.02314097 -0.1992535   0.09509248]
[[42. 42. 42. 42. 42. 42. 42. 42.]]
```

Custom Initialization

Sometimes, the initialization methods we need are not provided by the deep learning framework. In the example below, we define an initializer for any weight parameter w using the following strange distribution:

$$w \sim \begin{cases} U(5, 10) & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ U(-10, -5) & \text{with probability } \frac{1}{4} \end{cases} \tag{5.2.1}$$

Here we define a subclass of the `Initializer` class. Usually, we only need to implement the `_init_weight` function which takes a tensor argument (`data`) and assigns to it the desired initialized values.

```
class MyInit(init.Initializer):
    def _init_weight(self, name, data):
        print('Init', name, data.shape)
        data[:] = np.random.uniform(-10, 10, data.shape)
        data *= np.abs(data) >= 5
```

(continues on next page)

(continued from previous page)

```
net.initialize(MyInit(), force_reinit=True)
net[0].weight.data()[:2]
```

```
Init dense0_weight (8, 4)
Init dense1_weight (1, 8)
```

```
array([[ 0.       , -0.       , -0.       ,  8.522827 ],
       [ 0.       , -8.828651 , -0.       , -5.6012006]])
```

Note that we always have the option of setting parameters directly.

```
net[0].weight.data()[:] += 1
net[0].weight.data()[0, 0] = 42
net[0].weight.data()[0]
```

```
array([42.      ,  1.      ,  1.      ,  9.522827])
```

A note for advanced users: if you want to adjust parameters within an `autograd` scope, you need to use `set_data` to avoid confusing the automatic differentiation mechanics.

5.2.3 Tied Parameters

Often, we want to share parameters across multiple layers. Let us see how to do this elegantly. In the following we allocate a dense layer and then use its parameters specifically to set those of another layer.

```
net = nn.Sequential()
# We need to give the shared layer a name so that we can refer to its
# parameters
shared = nn.Dense(8, activation='relu')
net.add(nn.Dense(8, activation='relu'), shared,
        nn.Dense(8, activation='relu', params=shared.params), nn.Dense(10))
net.initialize()

X = np.random.uniform(size=(2, 20))
net(X)

# Check whether the parameters are the same
print(net[1].weight.data()[0] == net[2].weight.data()[0])
net[1].weight.data()[0, 0] = 100
# Make sure that they are actually the same object rather than just having the
# same value
print(net[1].weight.data()[0] == net[2].weight.data()[0])
```

```
[ True  True  True  True  True  True  True  True]
[ True  True  True  True  True  True  True  True]
```

This example shows that the parameters of the second and third layer are tied. They are not just equal, they are represented by the same exact tensor. Thus, if we change one of the parameters,

the other one changes, too. You might wonder, when parameters are tied what happens to the gradients? Since the model parameters contain gradients, the gradients of the second hidden layer and the third hidden layer are added together during backpropagation.

Summary

- We have several ways to access, initialize, and tie model parameters.
- We can use custom initialization.

Exercises

1. Use the `FancyMLP` model defined in Section 5.1 and access the parameters of the various layers.
2. Look at the initialization module document to explore different initializers.
3. Construct an MLP containing a shared parameter layer and train it. During the training process, observe the model parameters and gradients of each layer.
4. Why is sharing parameters a good idea?

Discussions[79]

5.3 Deferred Initialization

So far, it might seem that we got away with being sloppy in setting up our networks. Specifically, we did the following unintuitive things, which might not seem like they should work:

- We defined the network architectures without specifying the input dimensionality.
- We added layers without specifying the output dimension of the previous layer.
- We even "initialized" these parameters before providing enough information to determine how many parameters our models should contain.

You might be surprised that our code runs at all. After all, there is no way the deep learning framework could tell what the input dimensionality of a network would be. The trick here is that the framework *defers initialization*, waiting until the first time we pass data through the model, to infer the sizes of each layer on the fly.

Later on, when working with convolutional neural networks, this technique will become even more convenient since the input dimensionality (i.e., the resolution of an image) will affect the dimensionality of each subsequent layer. Hence, the ability to set parameters without the need to know, at the time of writing the code, what the dimensionality is can greatly simplify the task of specifying and subsequently modifying our models. Next, we go deeper into the mechanics of initialization.

[79] https://discuss.d2l.ai/t/56

5.3.1 Instantiating a Network

To begin, let us instantiate an MLP.

```
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()

def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(256, activation='relu'))
    net.add(nn.Dense(10))
    return net

net = get_net()
```

At this point, the network cannot possibly know the dimensions of the input layer's weights because the input dimension remains unknown. Consequently the framework has not yet initialized any parameters. We confirm by attempting to access the parameters below.

```
print(net.collect_params)
print(net.collect_params())
```

```
<bound method Block.collect_params of Sequential(
  (0): Dense(-1 -> 256, Activation(relu))
  (1): Dense(-1 -> 10, linear)
)>
sequential0_ (
  Parameter dense0_weight (shape=(256, -1), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, -1), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

Note that while the parameter objects exist, the input dimension to each layer is listed as -1. MXNet uses the special value -1 to indicate that the parameter dimension remains unknown. At this point, attempts to access `net[0].weight.data()` would trigger a runtime error stating that the network must be initialized before the parameters can be accessed. Now let us see what happens when we attempt to initialize parameters via the `initialize` function.

```
net.initialize()
net.collect_params()
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, -1), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, -1), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

As we can see, nothing has changed. When input dimensions are unknown, calls to initialize do not truly initialize the parameters. Instead, this call registers to MXNet that we wish (and option-

ally, according to which distribution) to initialize the parameters.

Next let us pass data through the network to make the framework finally initialize parameters.

```
X = np.random.uniform(size=(2, 20))
net(X)

net.collect_params()
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

As soon as we know the input dimensionality, 20, the framework can identify the shape of the first layer's weight matrix by plugging in the value of 20. Having recognized the first layer's shape, the framework proceeds to the second layer, and so on through the computational graph until all shapes are known. Note that in this case, only the first layer requires deferred initialization, but the framework initializes sequentially. Once all parameter shapes are known, the framework can finally initialize the parameters.

Summary

- Deferred initialization can be convenient, allowing the framework to infer parameter shapes automatically, making it easy to modify architectures and eliminating one common source of errors.
- We can pass data through the model to make the framework finally initialize parameters.

Exercises

1. What happens if you specify the input dimensions to the first layer but not to subsequent layers? Do you get immediate initialization?
2. What happens if you specify mismatching dimensions?
3. What would you need to do if you have input of varying dimensionality? Hint: look at the parameter tying.

Discussions[80]

[80] https://discuss.d2l.ai/t/280

5.4 Custom Layers

One factor behind deep learning's success is the availability of a wide range of layers that can be composed in creative ways to design architectures suitable for a wide variety of tasks. For instance, researchers have invented layers specifically for handling images, text, looping over sequential data, and performing dynamic programming. Sooner or later, you will encounter or invent a layer that does not exist yet in the deep learning framework. In these cases, you must build a custom layer. In this section, we show you how.

5.4.1 Layers without Parameters

To start, we construct a custom layer that does not have any parameters of its own. This should look familiar if you recall our introduction to block in Section 5.1. The following `CenteredLayer` class simply subtracts the mean from its input. To build it, we simply need to inherit from the base layer class and implement the forward propagation function.

```
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()

class CenteredLayer(nn.Block):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def forward(self, X):
        return X - X.mean()
```

Let us verify that our layer works as intended by feeding some data through it.

```
layer = CenteredLayer()
layer(np.array([1, 2, 3, 4, 5]))
```

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

We can now incorporate our layer as a component in constructing more complex models.

```
net = nn.Sequential()
net.add(nn.Dense(128), CenteredLayer())
net.initialize()
```

As an extra sanity check, we can send random data through the network and check that the mean is in fact 0. Because we are dealing with floating point numbers, we may still see a very small nonzero number due to quantization.

```
Y = net(np.random.uniform(size=(4, 8)))
Y.mean()
```

```
array(3.783498e-10)
```

5.4.2 Layers with Parameters

Now that we know how to define simple layers, let us move on to defining layers with parameters that can be adjusted through training. We can use built-in functions to create parameters, which provide some basic housekeeping functionality. In particular, they govern access, initialization, sharing, saving, and loading model parameters. This way, among other benefits, we will not need to write custom serialization routines for every custom layer.

Now let us implement our own version of the fully-connected layer. Recall that this layer requires two parameters, one to represent the weight and the other for the bias. In this implementation, we bake in the ReLU activation as a default. This layer requires to input arguments: `in_units` and `units`, which denote the number of inputs and outputs, respectively.

```
class MyDense(nn.Block):
    def __init__(self, units, in_units, **kwargs):
        super().__init__(**kwargs)
        self.weight = self.params.get('weight', shape=(in_units, units))
        self.bias = self.params.get('bias', shape=(units,))

    def forward(self, x):
        linear = np.dot(
            x, self.weight.data(ctx=x.ctx)) + self.bias.data(ctx=x.ctx)
        return npx.relu(linear)
```

Next, we instantiate the `MyDense` class and access its model parameters.

```
dense = MyDense(units=3, in_units=5)
dense.params
```

```
mydense0_ (
  Parameter mydense0_weight (shape=(5, 3), dtype=<class 'numpy.float32'>)
  Parameter mydense0_bias (shape=(3,), dtype=<class 'numpy.float32'>)
)
```

We can directly carry out forward propagation calculations using custom layers.

```
dense.initialize()
dense(np.random.uniform(size=(2, 5)))
```

```
array([[0.        , 0.01633355, 0.        ],
       [0.        , 0.01581812, 0.        ]])
```

We can also construct models using custom layers. Once we have that we can use it just like the built-in fully-connected layer.

```
net = nn.Sequential()
net.add(MyDense(8, in_units=64), MyDense(1, in_units=8))
net.initialize()
net(np.random.uniform(size=(2, 64)))
```

```
array([[0.06508517],
       [0.0615553 ]])
```

Summary

- We can design custom layers via the basic layer class. This allows us to define flexible new layers that behave differently from any existing layers in the library.
- Once defined, custom layers can be invoked in arbitrary contexts and architectures.
- Layers can have local parameters, which can be created through built-in functions.

Exercises

1. Design a layer that takes an input and computes a tensor reduction, i.e., it returns $y_k = \sum_{i,j} W_{ijk} x_i x_j$.
2. Design a layer that returns the leading half of the Fourier coefficients of the data.

Discussions[81]

5.5 File I/O

So far we discussed how to process data and how to build, train, and test deep learning models. However, at some point, we will hopefully be happy enough with the learned models that we will want to save the results for later use in various contexts (perhaps even to make predictions in deployment). Additionally, when running a long training process, the best practice is to periodically save intermediate results (checkpointing) to ensure that we do not lose several days worth of computation if we trip over the power cord of our server. Thus it is time to learn how to load and store both individual weight vectors and entire models. This section addresses both issues.

5.5.1 Loading and Saving Tensors

For individual tensors, we can directly invoke the `load` and `save` functions to read and write them respectively. Both functions require that we supply a name, and `save` requires as input the variable to be saved.

```
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()

x = np.arange(4)
npx.save('x-file', x)
```

We can now read the data from the stored file back into memory.

```
x2 = npx.load('x-file')
x2
```

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

[81] https://discuss.d2l.ai/t/58

We can store a list of tensors and read them back into memory.

```
y = np.zeros(4)
npx.save('x-files', [x, y])
x2, y2 = npx.load('x-files')
(x2, y2)
```

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

We can even write and read a dictionary that maps from strings to tensors. This is convenient when we want to read or write all the weights in a model.

```
mydict = {'x': x, 'y': y}
npx.save('mydict', mydict)
mydict2 = npx.load('mydict')
mydict2
```

```
{'x': array([0., 1., 2., 3.]), 'y': array([0., 0., 0., 0.])}
```

5.5.2 Loading and Saving Model Parameters

Saving individual weight vectors (or other tensors) is useful, but it gets very tedious if we want to save (and later load) an entire model. After all, we might have hundreds of parameter groups sprinkled throughout. For this reason the deep learning framework provides built-in functionalities to load and save entire networks. An important detail to note is that this saves model *parameters* and not the entire model. For example, if we have a 3-layer MLP, we need to specify the architecture separately. The reason for this is that the models themselves can contain arbitrary code, hence they cannot be serialized as naturally. Thus, in order to reinstate a model, we need to generate the architecture in code and then load the parameters from disk. Let us start with our familiar MLP.

```
class MLP(nn.Block):
    def __init__(self, **kwargs):
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu')
        self.output = nn.Dense(10)

    def forward(self, x):
        return self.output(self.hidden(x))

net = MLP()
net.initialize()
X = np.random.uniform(size=(2, 20))
Y = net(X)
```

Next, we store the parameters of the model as a file with the name “mlp.params”.

```
net.save_parameters('mlp.params')
```

To recover the model, we instantiate a clone of the original MLP model. Instead of randomly initializing the model parameters, we read the parameters stored in the file directly.

```
clone = MLP()
clone.load_parameters('mlp.params')
```

Since both instances have the same model parameters, the computational result of the same input `X` should be the same. Let us verify this.

```
Y_clone = clone(X)
Y_clone == Y
```

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

Summary

- The save and load functions can be used to perform file I/O for tensor objects.
- We can save and load the entire sets of parameters for a network via a parameter dictionary.
- Saving the architecture has to be done in code rather than in parameters.

Exercises

1. Even if there is no need to deploy trained models to a different device, what are the practical benefits of storing model parameters?
2. Assume that we want to reuse only parts of a network to be incorporated into a network of a different architecture. How would you go about using, say the first two layers from a previous network in a new network?
3. How would you go about saving the network architecture and parameters? What restrictions would you impose on the architecture?

Discussions[82]

5.6 GPUs

In Table 1.5.1, we discussed the rapid growth of computation over the past two decades. In a nutshell, GPU performance has increased by a factor of 1000 every decade since 2000. This offers great opportunities but it also suggests a significant need to provide such performance.

In this section, we begin to discuss how to harness this computational performance for your research. First by using single GPUs and at a later point, how to use multiple GPUs and multiple servers (with multiple GPUs).

Specifically, we will discuss how to use a single NVIDIA GPU for calculations. First, make sure you have at least one NVIDIA GPU installed. Then, download the NVIDIA driver and CUDA[83] and

[82] https://discuss.d2l.ai/t/60
[83] https://developer.nvidia.com/cuda-downloads

follow the prompts to set the appropriate path. Once these preparations are complete, the `nvidia-smi` command can be used to view the graphics card information.

```
!nvidia-smi
```

```
Fri Jun 25 05:42:18 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:1B.0 Off |                    0 |
| N/A   55C    P0    64W / 300W |  10341MiB / 16130MiB |     19%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:00:1C.0 Off |                    0 |
| N/A   60C    P0    71W / 300W |   8960MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:00:1D.0 Off |                    0 |
| N/A   51C    P0    68W / 300W |   6550MiB / 16130MiB |     19%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   51C    P0    69W / 300W |   6434MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      2624      C   ...conda3/envs/d2l-en-release-0/bin/python  1653MiB |
|    0     47319      C   ...buntu/miniconda3/envs/d2l-en/bin/python  2479MiB |
|    0     78495      C   ...buntu/miniconda3/envs/d2l-en/bin/python  2763MiB |
|    1     47319      C   ...buntu/miniconda3/envs/d2l-en/bin/python  2127MiB |
|    1     78495      C   ...buntu/miniconda3/envs/d2l-en/bin/python  2581MiB |
|    2      2624      C   ...conda3/envs/d2l-en-release-0/bin/python  1687MiB |
|    2     47319      C   ...buntu/miniconda3/envs/d2l-en/bin/python  2199MiB |
|    2     78495      C   ...buntu/miniconda3/envs/d2l-en/bin/python  2653MiB |
|    3     47319      C   ...buntu/miniconda3/envs/d2l-en/bin/python  2199MiB |
|    3     78495      C   ...buntu/miniconda3/envs/d2l-en/bin/python  2653MiB |
+-----------------------------------------------------------------------------+
```

You might have noticed that a MXNet tensor looks almost identical to a NumPy `ndarray`. But there are a few crucial differences. One of the key features that distinguishes MXNet from NumPy is its support for diverse hardware devices.

In MXNet, every array has a context. So far, by default, all variables and associated computation have been assigned to the CPU. Typically, other contexts might be various GPUs. Things can get even hairier when we deploy jobs across multiple servers. By assigning arrays to contexts intelligently, we can minimize the time spent transferring data between devices. For example, when training neural networks on a server with a GPU, we typically prefer for the model's parameters to live on the GPU.

Next, we need to confirm that the GPU version of MXNet is installed. If a CPU version of MXNet is already installed, we need to uninstall it first. For example, use the `pip uninstall mxnet` command, then install the corresponding MXNet version according to your CUDA version. Assuming you have CUDA 10.0 installed, you can install the MXNet version that supports CUDA 10.0 via `pip`

`install mxnet-cu100`.

To run the programs in this section, you need at least two GPUs. Note that this might be extravagant for most desktop computers but it is easily available in the cloud, e.g., by using the AWS EC2 multi-GPU instances. Almost all other sections do *not* require multiple GPUs. Instead, this is simply to illustrate how data flow between different devices.

5.6.1 Computing Devices

We can specify devices, such as CPUs and GPUs, for storage and calculation. By default, tensors are created in the main memory and then use the CPU to calculate it.

In MXNet, the CPU and GPU can be indicated by `cpu()` and `gpu()`. It should be noted that `cpu()` (or any integer in the parentheses) means all physical CPUs and memory. This means that MXNet's calculations will try to use all CPU cores. However, `gpu()` only represents one card and the corresponding memory. If there are multiple GPUs, we use `gpu(i)` to represent the i^{th} GPU (i starts from 0). Also, `gpu(0)` and `gpu()` are equivalent.

```
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()

npx.cpu(), npx.gpu(), npx.gpu(1)
```

```
(cpu(0), gpu(0), gpu(1))
```

We can query the number of available GPUs.

```
npx.num_gpus()
```

```
2
```

Now we define two convenient functions that allow us to run code even if the requested GPUs do not exist.

```
def try_gpu(i=0):  #@save
    """Return gpu(i) if exists, otherwise return cpu()."""
    return npx.gpu(i) if npx.num_gpus() >= i + 1 else npx.cpu()

def try_all_gpus():  #@save
    """Return all available GPUs, or [cpu()] if no GPU exists."""
    devices = [npx.gpu(i) for i in range(npx.num_gpus())]
    return devices if devices else [npx.cpu()]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(gpu(0), cpu(0), [gpu(0), gpu(1)])
```

5.6.2 Tensors and GPUs

By default, tensors are created on the CPU. We can query the device where the tensor is located.

```
x = np.array([1, 2, 3])
x.ctx
```

```
cpu(0)
```

It is important to note that whenever we want to operate on multiple terms, they need to be on the same device. For instance, if we sum two tensors, we need to make sure that both arguments live on the same device—otherwise the framework would not know where to store the result or even how to decide where to perform the computation.

Storage on the GPU

There are several ways to store a tensor on the GPU. For example, we can specify a storage device when creating a tensor. Next, we create the tensor variable `X` on the first `gpu`. The tensor created on a GPU only consumes the memory of this GPU. We can use the `nvidia-smi` command to view GPU memory usage. In general, we need to make sure that we do not create data that exceed the GPU memory limit.

```
X = np.ones((2, 3), ctx=try_gpu())
X
```

```
array([[1., 1., 1.],
       [1., 1., 1.]], ctx=gpu(0))
```

Assuming that you have at least two GPUs, the following code will create a random tensor on the second GPU.

```
Y = np.random.uniform(size=(2, 3), ctx=try_gpu(1))
Y
```

```
array([[0.67478997, 0.07540122, 0.9956977 ],
       [0.09488854, 0.415456  , 0.11231736]], ctx=gpu(1))
```

Copying

If we want to compute `X + Y`, we need to decide where to perform this operation. For instance, as shown in Fig. 5.6.1, we can transfer `X` to the second GPU and perform the operation there. *Do not* simply add `X` and `Y`, since this will result in an exception. The runtime engine would not know what to do: it cannot find data on the same device and it fails. Since `Y` lives on the second GPU, we need to move `X` there before we can add the two.

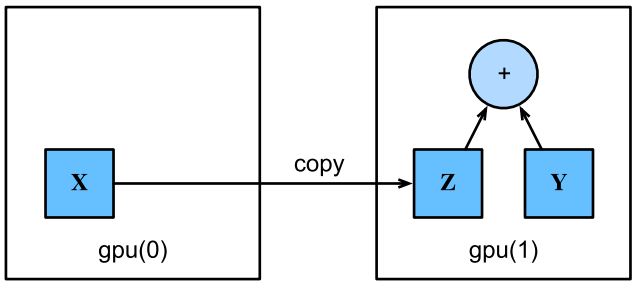


Fig. 5.6.1: Copy data to perform an operation on the same device.

```
Z = X.copyto(try_gpu(1))
print(X)
print(Z)
```

```
[[1. 1. 1.]
 [1. 1. 1.]] @gpu(0)
[[1. 1. 1.]
 [1. 1. 1.]] @gpu(1)
```

Now that the data are on the same GPU (both `Z` and `Y` are), we can add them up.

```
Y + Z
```

```
array([[1.6747899, 1.0754012, 1.9956977],
       [1.0948886, 1.415456 , 1.1123173]], ctx=gpu(1))
```

Imagine that your variable `Z` already lives on your second GPU. What happens if we still call `Z.copyto(gpu(1))`? It will make a copy and allocate new memory, even though that variable already lives on the desired device. There are times where, depending on the environment our code is running in, two variables may already live on the same device. So we want to make a copy only if the variables currently live in different devices. In these cases, we can call `as_in_ctx`. If the variable already live in the specified device then this is a no-op. Unless you specifically want to make a copy, `as_in_ctx` is the method of choice.

```
Z.as_in_ctx(try_gpu(1)) is Z
```

```
True
```

Side Notes

People use GPUs to do machine learning because they expect them to be fast. But transferring variables between devices is slow. So we want you to be 100% certain that you want to do something slow before we let you do it. If the deep learning framework just did the copy automatically without crashing then you might not realize that you had written some slow code.

Also, transferring data between devices (CPU, GPUs, and other machines) is something that is much slower than computation. It also makes parallelization a lot more difficult, since we have to wait for data to be sent (or rather to be received) before we can proceed with more operations. This

is why copy operations should be taken with great care. As a rule of thumb, many small operations are much worse than one big operation. Moreover, several operations at a time are much better than many single operations interspersed in the code unless you know what you are doing. This is the case since such operations can block if one device has to wait for the other before it can do something else. It is a bit like ordering your coffee in a queue rather than pre-ordering it by phone and finding out that it is ready when you are.

Last, when we print tensors or convert tensors to the NumPy format, if the data is not in the main memory, the framework will copy it to the main memory first, resulting in additional transmission overhead. Even worse, it is now subject to the dreaded global interpreter lock that makes everything wait for Python to complete.

5.6.3 Neural Networks and GPUs

Similarly, a neural network model can specify devices. The following code puts the model parameters on the GPU.

```
net = nn.Sequential()
net.add(nn.Dense(1))
net.initialize(ctx=try_gpu())
```

We will see many more examples of how to run models on GPUs in the following chapters, simply since they will become somewhat more computationally intensive.

When the input is a tensor on the GPU, the model will calculate the result on the same GPU.

```
net(X)
```

```
array([[0.04995865],
       [0.04995865]], ctx=gpu(0))
```

Let us confirm that the model parameters are stored on the same GPU.

```
net[0].weight.data().ctx
```

```
gpu(0)
```

In short, as long as all data and parameters are on the same device, we can learn models efficiently. In the following chapters we will see several such examples.

Summary

- We can specify devices for storage and calculation, such as the CPU or GPU. By default, data are created in the main memory and then use the CPU for calculations.
- The deep learning framework requires all input data for calculation to be on the same device, be it CPU or the same GPU.
- You can lose significant performance by moving data without care. A typical mistake is as follows: computing the loss for every minibatch on the GPU and reporting it back to the user on the command line (or logging it in a NumPy `ndarray`) will trigger a global interpreter lock

which stalls all GPUs. It is much better to allocate memory for logging inside the GPU and only move larger logs.

Exercises

1. Try a larger computation task, such as the multiplication of large matrices, and see the difference in speed between the CPU and GPU. What about a task with a small amount of calculations?
2. How should we read and write model parameters on the GPU?
3. Measure the time it takes to compute 1000 matrix-matrix multiplications of 100×100 matrices and log the Frobenius norm of the output matrix one result at a time vs. keeping a log on the GPU and transferring only the final result.
4. Measure how much time it takes to perform two matrix-matrix multiplications on two GPUs at the same time vs. in sequence on one GPU. Hint: you should see almost linear scaling.

Discussions[84]

[84] https://discuss.d2l.ai/t/62

6 | Convolutional Neural Networks

In earlier chapters, we came up against image data, for which each example consists of a two-dimensional grid of pixels. Depending on whether we are handling black-and-white or color images, each pixel location might be associated with either one or multiple numerical values, respectively. Until now, our way of dealing with this rich structure was deeply unsatisfying. We simply discarded each image's spatial structure by flattening them into one-dimensional vectors, feeding them through a fully-connected MLP. Because these networks are invariant to the order of the features, we could get similar results regardless of whether we preserve an order corresponding to the spatial structure of the pixels or if we permute the columns of our design matrix before fitting the MLP's parameters. Preferably, we would leverage our prior knowledge that nearby pixels are typically related to each other, to build efficient models for learning from image data.

This chapter introduces *convolutional neural networks* (CNNs), a powerful family of neural networks that are designed for precisely this purpose. CNN-based architectures are now ubiquitous in the field of computer vision, and have become so dominant that hardly anyone today would develop a commercial application or enter a competition related to image recognition, object detection, or semantic segmentation, without building off of this approach.

Modern CNNs, as they are called colloquially owe their design to inspirations from biology, group theory, and a healthy dose of experimental tinkering. In addition to their sample efficiency in achieving accurate models, CNNs tend to be computationally efficient, both because they require fewer parameters than fully-connected architectures and because convolutions are easy to parallelize across GPU cores. Consequently, practitioners often apply CNNs whenever possible, and increasingly they have emerged as credible competitors even on tasks with a one-dimensional sequence structure, such as audio, text, and time series analysis, where recurrent neural networks are conventionally used. Some clever adaptations of CNNs have also brought them to bear on graph-structured data and in recommender systems.

First, we will walk through the basic operations that comprise the backbone of all convolutional networks. These include the convolutional layers themselves, nitty-gritty details including padding and stride, the pooling layers used to aggregate information across adjacent spatial regions, the use of multiple channels at each layer, and a careful discussion of the structure of modern architectures. We will conclude the chapter with a full working example of LeNet, the first convolutional network successfully deployed, long before the rise of modern deep learning. In the next chapter, we will dive into full implementations of some popular and comparatively recent CNN architectures whose designs represent most of the techniques commonly used by modern practitioners.

6.1 From Fully-Connected Layers to Convolutions

To this day, the models that we have discussed so far remain appropriate options when we are dealing with tabular data. By tabular, we mean that the data consist of rows corresponding to examples and columns corresponding to features. With tabular data, we might anticipate that the patterns we seek could involve interactions among the features, but we do not assume any structure *a priori* concerning how the features interact.

Sometimes, we truly lack knowledge to guide the construction of craftier architectures. In these cases, an MLP may be the best that we can do. However, for high-dimensional perceptual data, such structure-less networks can grow unwieldy.

For instance, let us return to our running example of distinguishing cats from dogs. Say that we do a thorough job in data collection, collecting an annotated dataset of one-megapixel photographs. This means that each input to the network has one million dimensions. According to our discussions of parameterization cost of fully-connected layers in Section 3.4.3, even an aggressive reduction to one thousand hidden dimensions would require a fully-connected layer characterized by $10^6 \times 10^3 = 10^9$ parameters. Unless we have lots of GPUs, a talent for distributed optimization, and an extraordinary amount of patience, learning the parameters of this network may turn out to be infeasible.

A careful reader might object to this argument on the basis that one megapixel resolution may not be necessary. However, while we might be able to get away with one hundred thousand pixels, our hidden layer of size 1000 grossly underestimates the number of hidden units that it takes to learn good representations of images, so a practical system will still require billions of parameters. Moreover, learning a classifier by fitting so many parameters might require collecting an enormous dataset. And yet today both humans and computers are able to distinguish cats from dogs quite well, seemingly contradicting these intuitions. That is because images exhibit rich structure that can be exploited by humans and machine learning models alike. Convolutional neural networks (CNNs) are one creative way that machine learning has embraced for exploiting some of the known structure in natural images.

6.1.1 Invariance

Imagine that you want to detect an object in an image. It seems reasonable that whatever method we use to recognize objects should not be overly concerned with the precise location of the object in the image. Ideally, our system should exploit this knowledge. Pigs usually do not fly and planes usually do not swim. Nonetheless, we should still recognize a pig were one to appear at the top of the image. We can draw some inspiration here from the children's game "Where's Waldo" (depicted in Fig. 6.1.1). The game consists of a number of chaotic scenes bursting with activities. Waldo shows up somewhere in each, typically lurking in some unlikely location. The reader's goal is to locate him. Despite his characteristic outfit, this can be surprisingly difficult, due to the large number of distractions. However, *what Waldo looks like* does not depend upon *where Waldo is located*. We could sweep the image with a Waldo detector that could assign a score to each patch, indicating the likelihood that the patch contains Waldo. CNNs systematize this idea of *spatial invariance*, exploiting it to learn useful representations with fewer parameters.

Fig. 6.1.1: An image of the "Where's Waldo" game.

We can now make these intuitions more concrete by enumerating a few desiderata to guide our design of a neural network architecture suitable for computer vision:

1. In the earliest layers, our network should respond similarly to the same patch, regardless of where it appears in the image. This principle is called *translation invariance*.
2. The earliest layers of the network should focus on local regions, without regard for the contents of the image in distant regions. This is the *locality* principle. Eventually, these local representations can be aggregated to make predictions at the whole image level.

Let us see how this translates into mathematics.

6.1.2 Constraining the MLP

To start off, we can consider an MLP with two-dimensional images $\mathbf{X}$ as inputs and their immediate hidden representations $\mathbf{H}$ similarly represented as matrices in mathematics and as two-dimensional tensors in code, where both $\mathbf{X}$ and $\mathbf{H}$ have the same shape. Let that sink in. We now conceive of not only the inputs but also the hidden representations as possessing spatial structure.

Let $[\mathbf{X}]_{i,j}$ and $[\mathbf{H}]_{i,j}$ denote the pixel at location (i, j) in the input image and hidden representation, respectively. Consequently, to have each of the hidden units receive input from each of the input pixels, we would switch from using weight matrices (as we did previously in MLPs) to representing our parameters as fourth-order weight tensors W. Suppose that $\mathbf{U}$ contains biases, we could formally express the fully-connected layer as

$$\begin{aligned}[\mathbf{H}]_{i,j} &= [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l} \\ &= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}.\end{aligned}, \tag{6.1.1}$$

where the switch from W to V is entirely cosmetic for now since there is a one-to-one correspondence between coefficients in both fourth-order tensors. We simply re-index the subscripts (k, l) such that $k = i + a$ and $l = j + b$. In other words, we set $[\mathsf{V}]_{i,j,a,b} = [\mathsf{W}]_{i,j,i+a,j+b}$. The indices a and

b run over both positive and negative offsets, covering the entire image. For any given location (i, j) in the hidden representation $[\mathbf{H}]_{i,j}$, we compute its value by summing over pixels in x, centered around (i, j) and weighted by $[\mathsf{V}]_{i,j,a,b}$.

Translation Invariance

Now let us invoke the first principle established above: translation invariance. This implies that a shift in the input $\mathbf{X}$ should simply lead to a shift in the hidden representation $\mathbf{H}$. This is only possible if V and $\mathbf{U}$ do not actually depend on (i, j), i.e., we have $[\mathsf{V}]_{i,j,a,b} = [\mathbf{V}]_{a,b}$ and $\mathbf{U}$ is a constant, say u. As a result, we can simplify the definition for $\mathbf{H}$:

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{6.1.2}$$

This is a *convolution*! We are effectively weighting pixels at $(i + a, j + b)$ in the vicinity of location (i, j) with coefficients $[\mathbf{V}]_{a,b}$ to obtain the value $[\mathbf{H}]_{i,j}$. Note that $[\mathbf{V}]_{a,b}$ needs many fewer coefficients than $[\mathsf{V}]_{i,j,a,b}$ since it no longer depends on the location within the image. We have made significant progress!

Locality

Now let us invoke the second principle: locality. As motivated above, we believe that we should not have to look very far away from location (i, j) in order to glean relevant information to assess what is going on at $[\mathbf{H}]_{i,j}$. This means that outside some range $|a| > \Delta$ or $|b| > \Delta$, we should set $[\mathbf{V}]_{a,b} = 0$. Equivalently, we can rewrite $[\mathbf{H}]_{i,j}$ as

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{6.1.3}$$

Note that (6.1.3), in a nutshell, is a *convolutional layer*. *Convolutional neural networks* (CNNs) are a special family of neural networks that contain convolutional layers. In the deep learning research community, $\mathbf{V}$ is referred to as a *convolution kernel*, a *filter*, or simply the layer's *weights* that are often learnable parameters. When the local region is small, the difference as compared with a fully-connected network can be dramatic. While previously, we might have required billions of parameters to represent just a single layer in an image-processing network, we now typically need just a few hundred, without altering the dimensionality of either the inputs or the hidden representations. The price paid for this drastic reduction in parameters is that our features are now translation invariant and that our layer can only incorporate local information, when determining the value of each hidden activation. All learning depends on imposing inductive bias. When that bias agrees with reality, we get sample-efficient models that generalize well to unseen data. But of course, if those biases do not agree with reality, e.g., if images turned out not to be translation invariant, our models might struggle even to fit our training data.

6.1.3 Convolutions

Before going further, we should briefly review why the above operation is called a convolution. In mathematics, the *convolution* between two functions, say $f, g : \mathbb{R}^d \to \mathbb{R}$ is defined as

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z}) g(\mathbf{x} - \mathbf{z}) d\mathbf{z}. \tag{6.1.4}$$

That is, we measure the overlap between f and g when one function is "flipped" and shifted by $\mathbf{x}$. Whenever we have discrete objects, the integral turns into a sum. For instance, for vectors from the set of square summable infinite dimensional vectors with index running over $\mathbb{Z}$ we obtain the following definition:

$$(f * g)(i) = \sum_a f(a) g(i - a). \tag{6.1.5}$$

For two-dimensional tensors, we have a corresponding sum with indices (a, b) for f and $(i-a, j-b)$ for g, respectively:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b) g(i - a, j - b). \tag{6.1.6}$$

This looks similar to (6.1.3), with one major difference. Rather than using $(i+a, j+b)$, we are using the difference instead. Note, though, that this distinction is mostly cosmetic since we can always match the notation between (6.1.3) and (6.1.6). Our original definition in (6.1.3) more properly describes a *cross-correlation*. We will come back to this in the following section.

6.1.4 "Where's Waldo" Revisited

Returning to our Waldo detector, let us see what this looks like. The convolutional layer picks windows of a given size and weighs intensities according to the filter V, as demonstrated in Fig. 6.1.2. We might aim to learn a model so that wherever the "waldoness" is highest, we should find a peak in the hidden layer representations.

Fig. 6.1.2: Detect Waldo.

Channels

There is just one problem with this approach. So far, we blissfully ignored that images consist of 3 channels: red, green, and blue. In reality, images are not two-dimensional objects but rather third-order tensors, characterized by a height, width, and channel, e.g., with shape $1024 \times 1024 \times 3$ pixels. While the first two of these axes concern spatial relationships, the third can be regarded as assigning a multidimensional representation to each pixel location. We thus index X as $[\mathsf{X}]_{i,j,k}$. The convolutional filter has to adapt accordingly. Instead of $[\mathbf{V}]_{a,b}$, we now have $[\mathsf{V}]_{a,b,c}$.

Moreover, just as our input consists of a third-order tensor, it turns out to be a good idea to similarly formulate our hidden representations as third-order tensors H. In other words, rather than just having a single hidden representation corresponding to each spatial location, we want an entire vector of hidden representations corresponding to each spatial location. We could think of the hidden representations as comprising a number of two-dimensional grids stacked on top of each other. As in the inputs, these are sometimes called *channels*. They are also sometimes called *feature maps*, as each provides a spatialized set of learned features to the subsequent layer. Intuitively, you might imagine that at lower layers that are closer to inputs, some channels could become specialized to recognize edges while others could recognize textures.

To support multiple channels in both inputs (X) and hidden representations (H), we can add a fourth coordinate to V: $[\mathsf{V}]_{a,b,c,d}$. Putting everything together we have:

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} [\mathsf{V}]_{a,b,c,d} [\mathsf{X}]_{i+a,j+b,c}, \tag{6.1.7}$$

where d indexes the output channels in the hidden representations H. The subsequent convolutional layer will go on to take a third-order tensor, H, as the input. Being more general, (6.1.7) is the definition of a convolutional layer for multiple channels, where V is a kernel or filter of the layer.

There are still many operations that we need to address. For instance, we need to figure out how to combine all the hidden representations to a single output, e.g., whether there is a Waldo *anywhere* in the image. We also need to decide how to compute things efficiently, how to combine multiple layers, appropriate activation functions, and how to make reasonable design choices to yield networks that are effective in practice. We turn to these issues in the remainder of the chapter.

Summary

- Translation invariance in images implies that all patches of an image will be treated in the same manner.
- Locality means that only a small neighborhood of pixels will be used to compute the corresponding hidden representations.
- In image processing, convolutional layers typically require many fewer parameters than fully-connected layers.
- CNNS are a special family of neural networks that contain convolutional layers.
- Channels on input and output allow our model to capture multiple aspects of an image at each spatial location.

Exercises

1. Assume that the size of the convolution kernel is $\Delta = 0$. Show that in this case the convolution kernel implements an MLP independently for each set of channels.
2. Why might translation invariance not be a good idea after all?
3. What problems must we deal with when deciding how to treat hidden representations corresponding to pixel locations at the boundary of an image?
4. Describe an analogous convolutional layer for audio.
5. Do you think that convolutional layers might also be applicable for text data? Why or why not?
6. Prove that $f * g = g * f$.

Discussions[85]

6.2 Convolutions for Images

Now that we understand how convolutional layers work in theory, we are ready to see how they work in practice. Building on our motivation of convolutional neural networks as efficient architectures for exploring structure in image data, we stick with images as our running example.

6.2.1 The Cross-Correlation Operation

Recall that strictly speaking, convolutional layers are a misnomer, since the operations they express are more accurately described as cross-correlations. Based on our descriptions of convolutional layers in Section 6.1, in such a layer, an input tensor and a kernel tensor are combined to produce an output tensor through a cross-correlation operation.

Let us ignore channels for now and see how this works with two-dimensional data and hidden representations. In Fig. 6.2.1, the input is a two-dimensional tensor with a height of 3 and width of 3. We mark the shape of the tensor as 3×3 or $(3, 3)$. The height and width of the kernel are both 2. The shape of the *kernel window* (or *convolution window*) is given by the height and width of the kernel (here it is 2×2).

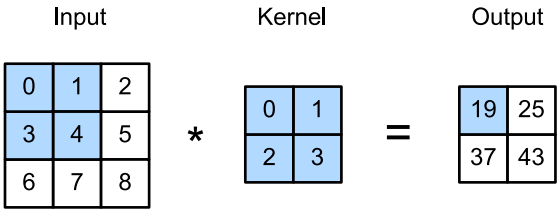


Fig. 6.2.1: Two-dimensional cross-correlation operation. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

In the two-dimensional cross-correlation operation, we begin with the convolution window positioned at the upper-left corner of the input tensor and slide it across the input tensor, both from

[85] https://discuss.d2l.ai/t/64

left to right and top to bottom. When the convolution window slides to a certain position, the input subtensor contained in that window and the kernel tensor are multiplied elementwise and the resulting tensor is summed up yielding a single scalar value. This result gives the value of the output tensor at the corresponding location. Here, the output tensor has a height of 2 and width of 2 and the four elements are derived from the two-dimensional cross-correlation operation:

$$\begin{aligned}0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,\\ 1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,\\ 3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,\\ 4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.\end{aligned} \tag{6.2.1}$$

Note that along each axis, the output size is slightly smaller than the input size. Because the kernel has width and height greater than one, we can only properly compute the cross-correlation for locations where the kernel fits wholly within the image, the output size is given by the input size $n_h \times n_w$ minus the size of the convolution kernel $k_h \times k_w$ via

$$(n_h - k_h + 1) \times (n_w - k_w + 1). \tag{6.2.2}$$

This is the case since we need enough space to "shift" the convolution kernel across the image. Later we will see how to keep the size unchanged by padding the image with zeros around its boundary so that there is enough space to shift the kernel. Next, we implement this process in the `corr2d` function, which accepts an input tensor `X` and a kernel tensor `K` and returns an output tensor `Y`.

```
from mxnet import autograd, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

```
def corr2d(X, K):  #@save
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = np.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y
```

We can construct the input tensor `X` and the kernel tensor `K` from Fig. 6.2.1 to validate the output of the above implementation of the two-dimensional cross-correlation operation.

```
X = np.array([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = np.array([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

```
array([[19., 25.],
       [37., 43.]])
```

6.2.2 Convolutional Layers

A convolutional layer cross-correlates the input and kernel and adds a scalar bias to produce an output. The two parameters of a convolutional layer are the kernel and the scalar bias. When training models based on convolutional layers, we typically initialize the kernels randomly, just as we would with a fully-connected layer.

We are now ready to implement a two-dimensional convolutional layer based on the `corr2d` function defined above. In the `__init__` constructor function, we declare `weight` and `bias` as the two model parameters. The forward propagation function calls the `corr2d` function and adds the bias.

```
class Conv2D(nn.Block):
    def __init__(self, kernel_size, **kwargs):
        super().__init__(**kwargs)
        self.weight = self.params.get('weight', shape=kernel_size)
        self.bias = self.params.get('bias', shape=(1,))

    def forward(self, x):
        return corr2d(x, self.weight.data()) + self.bias.data()
```

In $h \times w$ convolution or a $h \times w$ convolution kernel, the height and width of the convolution kernel are h and w, respectively. We also refer to a convolutional layer with a $h \times w$ convolution kernel simply as a $h \times w$ convolutional layer.

6.2.3 Object Edge Detection in Images

Let us take a moment to parse a simple application of a convolutional layer: detecting the edge of an object in an image by finding the location of the pixel change. First, we construct an "image" of 6×8 pixels. The middle four columns are black (0) and the rest are white (1).

```
X = np.ones((6, 8))
X[:, 2:6] = 0
X
```

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

Next, we construct a kernel `K` with a height of 1 and a width of 2. When we perform the cross-correlation operation with the input, if the horizontally adjacent elements are the same, the output is 0. Otherwise, the output is non-zero.

```
K = np.array([[1.0, -1.0]])
```

We are ready to perform the cross-correlation operation with arguments `X` (our input) and `K` (our kernel). As you can see, we detect 1 for the edge from white to black and -1 for the edge from black to white. All other outputs take value 0.

```
Y = corr2d(X, K)
Y
```

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

We can now apply the kernel to the transposed image. As expected, it vanishes. The kernel `K` only detects vertical edges.

```
corr2d(X.T, K)
```

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

6.2.4 Learning a Kernel

Designing an edge detector by finite differences `[1, -1]` is neat if we know this is precisely what we are looking for. However, as we look at larger kernels, and consider successive layers of convolutions, it might be impossible to specify precisely what each filter should be doing manually.

Now let us see whether we can learn the kernel that generated `Y` from `X` by looking at the input–output pairs only. We first construct a convolutional layer and initialize its kernel as a random tensor. Next, in each iteration, we will use the squared error to compare `Y` with the output of the convolutional layer. We can then calculate the gradient to update the kernel. For the sake of simplicity, in the following we use the built-in class for two-dimensional convolutional layers and ignore the bias.

```
# Construct a two-dimensional convolutional layer with 1 output channel and a
# kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
conv2d = nn.Conv2D(1, kernel_size=(1, 2), use_bias=False)
conv2d.initialize()

# The two-dimensional convolutional layer uses four-dimensional input and
# output in the format of (example, channel, height, width), where the batch
# size (number of examples in the batch) and the number of channels are both 1
X = X.reshape(1, 1, 6, 8)
Y = Y.reshape(1, 1, 6, 7)
lr = 3e-2  # Learning rate

for i in range(10):
    with autograd.record():
```

(continues on next page)

(continued from previous page)

```
        Y_hat = conv2d(X)
        l = (Y_hat - Y)**2
    l.backward()
    # Update the kernel
    conv2d.weight.data()[:] -= lr * conv2d.weight.grad()
    if (i + 1) % 2 == 0:
        print(f'batch {i + 1}, loss {float(l.sum()):.3f}')
```

```
batch 2, loss 4.949
batch 4, loss 0.831
batch 6, loss 0.140
batch 8, loss 0.024
batch 10, loss 0.004
```

Note that the error has dropped to a small value after 10 iterations. Now we will take a look at the kernel tensor we learned.

```
conv2d.weight.data().reshape((1, 2))
```

```
array([[ 0.9895   , -0.9873705]])
```

Indeed, the learned kernel tensor is remarkably close to the kernel tensor `K` we defined earlier.

6.2.5 Cross-Correlation and Convolution

Recall our observation from Section 6.1 of the correspondence between the cross-correlation and convolution operations. Here let us continue to consider two-dimensional convolutional layers. What if such layers perform strict convolution operations as defined in (6.1.6) instead of cross-correlations? In order to obtain the output of the strict *convolution* operation, we only need to flip the two-dimensional kernel tensor both horizontally and vertically, and then perform the *cross-correlation* operation with the input tensor.

It is noteworthy that since kernels are learned from data in deep learning, the outputs of convolutional layers remain unaffected no matter such layers perform either the strict convolution operations or the cross-correlation operations.

To illustrate this, suppose that a convolutional layer performs *cross-correlation* and learns the kernel in Fig. 6.2.1, which is denoted as the matrix $\mathbf{K}$ here. Assuming that other conditions remain unchanged, when this layer performs strict *convolution* instead, the learned kernel $\mathbf{K}'$ will be the same as $\mathbf{K}$ after $\mathbf{K}'$ is flipped both horizontally and vertically. That is to say, when the convolutional layer performs strict *convolution* for the input in Fig. 6.2.1 and $\mathbf{K}'$, the same output in Fig. 6.2.1 (cross-correlation of the input and $\mathbf{K}$) will be obtained.

In keeping with standard terminology with deep learning literature, we will continue to refer to the cross-correlation operation as a convolution even though, strictly-speaking, it is slightly different. Besides, we use the term *element* to refer to an entry (or component) of any tensor representing a layer representation or a convolution kernel.

6.2.6 Feature Map and Receptive Field

As described in Section 6.1.4, the convolutional layer output in Fig. 6.2.1 is sometimes called a *feature map*, as it can be regarded as the learned representations (features) in the spatial dimensions (e.g., width and height) to the subsequent layer. In CNNs, for any element x of some layer, its *receptive field* refers to all the elements (from all the previous layers) that may affect the calculation of x during the forward propagation. Note that the receptive field may be larger than the actual size of the input.

Let us continue to use Fig. 6.2.1 to explain the receptive field. Given the 2×2 convolution kernel, the receptive field of the shaded output element (of value 19) is the four elements in the shaded portion of the input. Now let us denote the 2×2 output as $\mathbf{Y}$ and consider a deeper CNN with an additional 2×2 convolutional layer that takes $\mathbf{Y}$ as its input, outputting a single element z. In this case, the receptive field of z on $\mathbf{Y}$ includes all the four elements of $\mathbf{Y}$, while the receptive field on the input includes all the nine input elements. Thus, when any element in a feature map needs a larger receptive field to detect input features over a broader area, we can build a deeper network.

Summary

- The core computation of a two-dimensional convolutional layer is a two-dimensional cross-correlation operation. In its simplest form, this performs a cross-correlation operation on the two-dimensional input data and the kernel, and then adds a bias.
- We can design a kernel to detect edges in images.
- We can learn the kernel's parameters from data.
- With kernels learned from data, the outputs of convolutional layers remain unaffected regardless of such layers' performed operations (either strict convolution or cross-correlation).
- When any element in a feature map needs a larger receptive field to detect broader features on the input, a deeper network can be considered.

Exercises

1. Construct an image `X` with diagonal edges.
    1. What happens if you apply the kernel `K` in this section to it?
    2. What happens if you transpose `X`?
    3. What happens if you transpose `K`?
2. When you try to automatically find the gradient for the `Conv2D` class we created, what kind of error message do you see?
3. How do you represent a cross-correlation operation as a matrix multiplication by changing the input and kernel tensors?
4. Design some kernels manually.
    1. What is the form of a kernel for the second derivative?
    2. What is the kernel for an integral?

3. What is the minimum size of a kernel to obtain a derivative of degree d?

Discussions[86]

6.3 Padding and Stride

In the previous example of Fig. 6.2.1, our input had both a height and width of 3 and our convolution kernel had both a height and width of 2, yielding an output representation with dimension 2×2. As we generalized in Section 6.2, assuming that the input shape is $n_h \times n_w$ and the convolution kernel shape is $k_h \times k_w$, then the output shape will be $(n_h - k_h + 1) \times (n_w - k_w + 1)$. Therefore, the output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel.

In several cases, we incorporate techniques, including padding and strided convolutions, that affect the size of the output. As motivation, note that since kernels generally have width and height greater than 1, after applying many successive convolutions, we tend to wind up with outputs that are considerably smaller than our input. If we start with a 240×240 pixel image, 10 layers of 5×5 convolutions reduce the image to 200×200 pixels, slicing off 30% of the image and with it obliterating any interesting information on the boundaries of the original image. *Padding* is the most popular tool for handling this issue.

In other cases, we may want to reduce the dimensionality drastically, e.g., if we find the original input resolution to be unwieldy. *Strided convolutions* are a popular technique that can help in these instances.

6.3.1 Padding

As described above, one tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image. Since we typically use small kernels, for any given convolution, we might only lose a few pixels, but this can add up as we apply many successive convolutional layers. One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. Typically, we set the values of the extra pixels to zero. In Fig. 6.3.1, we pad a 3×3 input, increasing its size to 5×5. The corresponding output then increases to a 4×4 matrix. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

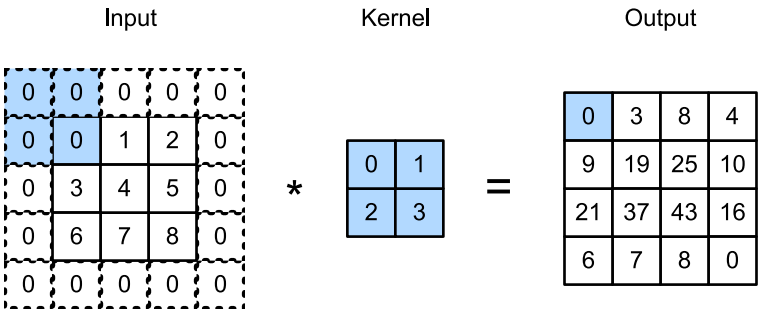


Fig. 6.3.1: Two-dimensional cross-correlation with padding.

[86] https://discuss.d2l.ai/t/65

In general, if we add a total of p_h rows of padding (roughly half on top and half on bottom) and a total of p_w columns of padding (roughly half on the left and half on the right), the output shape will be

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1). \tag{6.3.1}$$

This means that the height and width of the output will increase by p_h and p_w, respectively.

In many cases, we will want to set $p_h = k_h - 1$ and $p_w = k_w - 1$ to give the input and output the same height and width. This will make it easier to predict the output shape of each layer when constructing the network. Assuming that k_h is odd here, we will pad $p_h/2$ rows on both sides of the height. If k_h is even, one possibility is to pad $\lceil p_h/2 \rceil$ rows on the top of the input and $\lfloor p_h/2 \rfloor$ rows on the bottom. We will pad both sides of the width in the same way.

CNNs commonly use convolution kernels with odd height and width values, such as 1, 3, 5, or 7. Choosing odd kernel sizes has the benefit that we can preserve the spatial dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right.

Moreover, this practice of using odd kernels and padding to precisely preserve dimensionality offers a clerical benefit. For any two-dimensional tensor `X`, when the kernel's size is odd and the number of padding rows and columns on all sides are the same, producing an output with the same height and width as the input, we know that the output `Y[i, j]` is calculated by cross-correlation of the input and convolution kernel with the window centered on `X[i, j]`.

In the following example, we create a two-dimensional convolutional layer with a height and width of 3 and apply 1 pixel of padding on all sides. Given an input with a height and width of 8, we find that the height and width of the output is also 8.

```
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()

# For convenience, we define a function to calculate the convolutional layer.
# This function initializes the convolutional layer weights and performs
# corresponding dimensionality elevations and reductions on the input and
# output
def comp_conv2d(conv2d, X):
    conv2d.initialize()
    # Here (1, 1) indicates that the batch size and the number of channels
    # are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Exclude the first two dimensions that do not interest us: examples and
    # channels
    return Y.reshape(Y.shape[2:])

# Note that here 1 row or column is padded on either side, so a total of 2
# rows or columns are added
conv2d = nn.Conv2D(1, kernel_size=3, padding=1)
X = np.random.uniform(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

When the height and width of the convolution kernel are different, we can make the output and input have the same height and width by setting different padding numbers for height and width.

```
# Here, we use a convolution kernel with a height of 5 and a width of 3. The
# padding numbers on either side of the height and width are 2 and 1,
# respectively
conv2d = nn.Conv2D(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

6.3.2 Stride

When computing the cross-correlation, we start with the convolution window at the upper-left corner of the input tensor, and then slide it over all locations both down and to the right. In previous examples, we default to sliding one element at a time. However, sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one element at a time, skipping the intermediate locations.

We refer to the number of rows and columns traversed per slide as the *stride*. So far, we have used strides of 1, both for height and width. Sometimes, we may want to use a larger stride. Fig. 6.3.2 shows a two-dimensional cross-correlation operation with a stride of 3 vertically and 2 horizontally. The shaded portions are the output elements as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$, $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$. We can see that when the second element of the first column is outputted, the convolution window slides down three rows. The convolution window slides two columns to the right when the second element of the first row is outputted. When the convolution window continues to slide two columns to the right on the input, there is no output because the input element cannot fill the window (unless we add another column of padding).

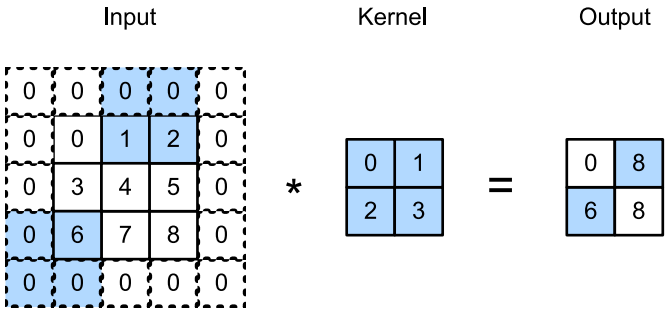


Fig. 6.3.2: Cross-correlation with strides of 3 and 2 for height and width, respectively.

In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor. \tag{6.3.2}$$

If we set $p_h = k_h - 1$ and $p_w = k_w - 1$, then the output shape will be simplified to $\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$. Going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be $(n_h/s_h) \times (n_w/s_w)$.

Below, we set the strides on both the height and width to 2, thus halving the input height and width.

```
conv2d = nn.Conv2D(1, kernel_size=3, padding=1, strides=2)
comp_conv2d(conv2d, X).shape
```

```
(4, 4)
```

Next, we will look at a slightly more complicated example.

```
conv2d = nn.Conv2D(1, kernel_size=(3, 5), padding=(0, 1), strides=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
(2, 2)
```

For the sake of brevity, when the padding number on both sides of the input height and width are p_h and p_w respectively, we call the padding (p_h, p_w). Specifically, when $p_h = p_w = p$, the padding is p. When the strides on the height and width are s_h and s_w, respectively, we call the stride (s_h, s_w). Specifically, when $s_h = s_w = s$, the stride is s. By default, the padding is 0 and the stride is 1. In practice, we rarely use inhomogeneous strides or padding, i.e., we usually have $p_h = p_w$ and $s_h = s_w$.

Summary

- Padding can increase the height and width of the output. This is often used to give the output the same height and width as the input.
- The stride can reduce the resolution of the output, for example reducing the height and width of the output to only $1/n$ of the height and width of the input (n is an integer greater than 1).
- Padding and stride can be used to adjust the dimensionality of the data effectively.

Exercises

1. For the last example in this section, use mathematics to calculate the output shape to see if it is consistent with the experimental result.
2. Try other padding and stride combinations on the experiments in this section.
3. For audio signals, what does a stride of 2 correspond to?
4. What are the computational benefits of a stride larger than 1?

Discussions[87]

[87] https://discuss.d2l.ai/t/67

6.4 Multiple Input and Multiple Output Channels

While we have described the multiple channels that comprise each image (e.g., color images have the standard RGB channels to indicate the amount of red, green and blue) and convolutional layers for multiple channels in Section 6.1.4, until now, we simplified all of our numerical examples by working with just a single input and a single output channel. This has allowed us to think of our inputs, convolution kernels, and outputs each as two-dimensional tensors.

When we add channels into the mix, our inputs and hidden representations both become three-dimensional tensors. For example, each RGB input image has shape $3 \times h \times w$. We refer to this axis, with a size of 3, as the *channel* dimension. In this section, we will take a deeper look at convolution kernels with multiple input and multiple output channels.

6.4.1 Multiple Input Channels

When the input data contain multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data. Assuming that the number of channels for the input data is c_i, the number of input channels of the convolution kernel also needs to be c_i. If our convolution kernel's window shape is $k_h \times k_w$, then when $c_i = 1$, we can think of our convolution kernel as just a two-dimensional tensor of shape $k_h \times k_w$.

However, when $c_i > 1$, we need a kernel that contains a tensor of shape $k_h \times k_w$ for *every* input channel. Concatenating these c_i tensors together yields a convolution kernel of shape $c_i \times k_h \times k_w$. Since the input and convolution kernel each have c_i channels, we can perform a cross-correlation operation on the two-dimensional tensor of the input and the two-dimensional tensor of the convolution kernel for each channel, adding the c_i results together (summing over the channels) to yield a two-dimensional tensor. This is the result of a two-dimensional cross-correlation between a multi-channel input and a multi-input-channel convolution kernel.

In Fig. 6.4.1, we demonstrate an example of a two-dimensional cross-correlation with two input channels. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $(1\times1+2\times2+4\times3+5\times4)+(0\times0+1\times1+3\times2+4\times3) = 56$.

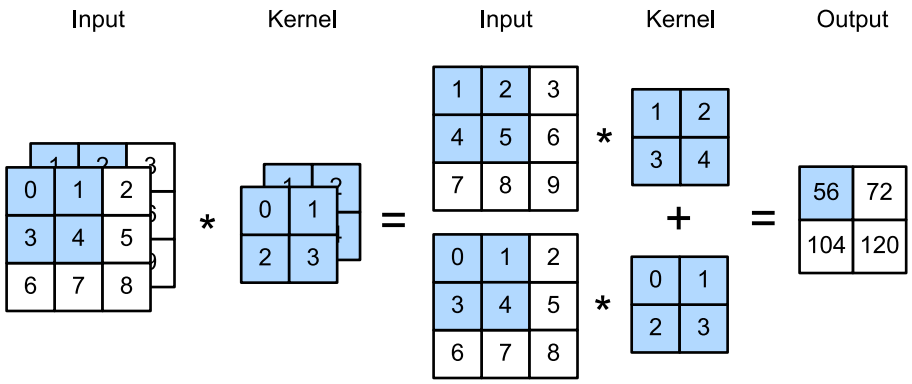


Fig. 6.4.1: Cross-correlation computation with 2 input channels.

To make sure we really understand what is going on here, we can implement cross-correlation operations with multiple input channels ourselves. Notice that all we are doing is performing one cross-correlation operation per channel and then adding up the results.

```
from mxnet import np, npx
from d2l import mxnet as d2l

npx.set_np()
```

```
def corr2d_multi_in(X, K):
    # First, iterate through the 0th dimension (channel dimension) of `X` and
    # `K`. Then, add them together
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

We can construct the input tensor `X` and the kernel tensor `K` corresponding to the values in Fig. 6.4.1 to validate the output of the cross-correlation operation.

```
X = np.array([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
              [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = np.array([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

```
array([[ 56.,  72.],
       [104., 120.]])
```

6.4.2 Multiple Output Channels

Regardless of the number of input channels, so far we always ended up with one output channel. However, as we discussed in Section 6.1.4, it turns out to be essential to have multiple channels at each layer. In the most popular neural network architectures, we actually increase the channel dimension as we go higher up in the neural network, typically downsampling to trade off spatial resolution for greater *channel depth*. Intuitively, you could think of each channel as responding to some different set of features. Reality is a bit more complicated than the most naive interpretations of this intuition since representations are not learned independent but are rather optimized to be jointly useful. So it may not be that a single channel learns an edge detector but rather that some direction in channel space corresponds to detecting edges.

Denote by c_i and c_o the number of input and output channels, respectively, and let k_h and k_w be the height and width of the kernel. To get an output with multiple channels, we can create a kernel tensor of shape $c_i \times k_h \times k_w$ for *every* output channel. We concatenate them on the output channel dimension, so that the shape of the convolution kernel is $c_o \times c_i \times k_h \times k_w$. In cross-correlation operations, the result on each output channel is calculated from the convolution kernel corresponding to that output channel and takes input from all channels in the input tensor.

We implement a cross-correlation function to calculate the output of multiple channels as shown below.

```
def corr2d_multi_in_out(X, K):
    # Iterate through the 0th dimension of `K`, and each time, perform
    # cross-correlation operations with input `X`. All of the results are
    # stacked together
    return np.stack([corr2d_multi_in(X, k) for k in K], 0)
```

We construct a convolution kernel with 3 output channels by concatenating the kernel tensor `K` with `K+1` (plus one for each element in `K`) and `K+2`.

```
K = np.stack((K, K + 1, K + 2), 0)
K.shape
```

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

Below, we perform cross-correlation operations on the input tensor `X` with the kernel tensor `K`. Now the output contains 3 channels. The result of the first channel is consistent with the result of the previous input tensor `X` and the multi-input channel, single-output channel kernel.

```
corr2d_multi_in_out(X, K)
```

```
array([[[ 56.,  72.],
        [104., 120.]],

       [[ 76., 100.],
        [148., 172.]],

       [[ 96., 128.],
        [192., 224.]]])
```

6.4.3 1×1 Convolutional Layer

At first, a 1×1 convolution, i.e., $k_h = k_w = 1$, does not seem to make much sense. After all, a convolution correlates adjacent pixels. A 1×1 convolution obviously does not. Nonetheless, they are popular operations that are sometimes included in the designs of complex deep networks. Let us see in some detail what it actually does.

Because the minimum window is used, the 1×1 convolution loses the ability of larger convolutional layers to recognize patterns consisting of interactions among adjacent elements in the height and width dimensions. The only computation of the 1×1 convolution occurs on the channel dimension.

Fig. 6.4.2 shows the cross-correlation computation using the 1×1 convolution kernel with 3 input channels and 2 output channels. Note that the inputs and outputs have the same height and width. Each element in the output is derived from a linear combination of elements *at the same position* in the input image. You could think of the 1×1 convolutional layer as constituting a fully-connected layer applied at every single pixel location to transform the c_i corresponding input values into c_o output values. Because this is still a convolutional layer, the weights are tied across pixel location. Thus the 1×1 convolutional layer requires $c_o \times c_i$ weights (plus the bias).

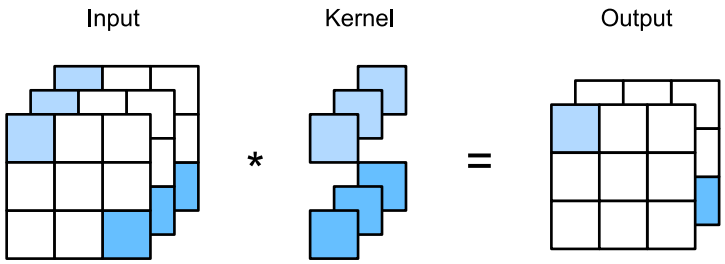


Fig. 6.4.2: The cross-correlation computation uses the 1×1 convolution kernel with 3 input channels and 2 output channels. The input and output have the same height and width.

Let us check whether this works in practice: we implement a 1×1 convolution using a fully-connected layer. The only thing is that we need to make some adjustments to the data shape before and after the matrix multiplication.

```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.reshape((c_i, h * w))
    K = K.reshape((c_o, c_i))
    # Matrix multiplication in the fully-connected layer
    Y = np.dot(K, X)
    return Y.reshape((c_o, h, w))
```

When performing 1×1 convolution, the above function is equivalent to the previously implemented cross-correlation function `corr2d_multi_in_out`. Let us check this with some sample data.

```
X = np.random.normal(0, 1, (3, 3, 3))
K = np.random.normal(0, 1, (2, 3, 1, 1))
```

```
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(np.abs(Y1 - Y2).sum()) < 1e-6
```

Summary

- Multiple channels can be used to extend the model parameters of the convolutional layer.
- The 1×1 convolutional layer is equivalent to the fully-connected layer, when applied on a per pixel basis.
- The 1×1 convolutional layer is typically used to adjust the number of channels between network layers and to control model complexity.

Exercises

1. Assume that we have two convolution kernels of size k_1 and k_2, respectively (with no nonlinearity in between).
    1. Prove that the result of the operation can be expressed by a single convolution.
    2. What is the dimensionality of the equivalent single convolution?
    3. Is the converse true?
2. Assume an input of shape $c_i \times h \times w$ and a convolution kernel of shape $c_o \times c_i \times k_h \times k_w$, padding of (p_h, p_w), and stride of (s_h, s_w).
    1. What is the computational cost (multiplications and additions) for the forward propagation?
    2. What is the memory footprint?
    3. What is the memory footprint for the backward computation?
    4. What is the computational cost for the backpropagation?
3. By what factor does the number of calculations increase if we double the number of input channels c_i and the number of output channels c_o? What happens if we double the padding?
4. If the height and width of a convolution kernel is $k_h = k_w = 1$, what is the computational complexity of the forward propagation?
5. Are the variables `Y1` and `Y2` in the last example of this section exactly the same? Why?
6. How would you implement convolutions using matrix multiplication when the convolution window is not 1×1?

Discussions[88]

6.5 Pooling

Often, as we process images, we want to gradually reduce the spatial resolution of our hidden representations, aggregating information so that the higher up we go in the network, the larger the receptive field (in the input) to which each hidden node is sensitive.

Often our ultimate task asks some global question about the image, e.g., *does it contain a cat?* So typically the units of our final layer should be sensitive to the entire input. By gradually aggregating information, yielding coarser and coarser maps, we accomplish this goal of ultimately learning a global representation, while keeping all of the advantages of convolutional layers at the intermediate layers of processing.

Moreover, when detecting lower-level features, such as edges (as discussed in Section 6.2), we often want our representations to be somewhat invariant to translation. For instance, if we take the image `X` with a sharp delineation between black and white and shift the whole image by one pixel to the right, i.e., `Z[i, j] = X[i, j + 1]`, then the output for the new image `Z` might be vastly different. The edge will have shifted by one pixel. In reality, objects hardly ever occur exactly at the same place. In fact, even with a tripod and a stationary object, vibration of the camera due to

[88] https://discuss.d2l.ai/t/69

the movement of the shutter might shift everything by a pixel or so (high-end cameras are loaded with special features to address this problem).

This section introduces *pooling layers*, which serve the dual purposes of mitigating the sensitivity of convolutional layers to location and of spatially downsampling representations.

6.5.1 Maximum Pooling and Average Pooling

Like convolutional layers, *pooling* operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the *pooling window*). However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters (there is no *kernel*). Instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window. These operations are called *maximum pooling* (*max pooling* for short) and *average pooling*, respectively.

In both cases, as with the cross-correlation operator, we can think of the pooling window as starting from the upper-left of the input tensor and sliding across the input tensor from left to right and top to bottom. At each location that the pooling window hits, it computes the maximum or average value of the input subtensor in the window, depending on whether max or average pooling is employed.

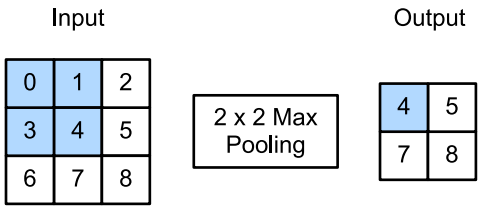


Fig. 6.5.1: Maximum pooling with a pooling window shape of 2×2. The shaded portions are the first output element as well as the input tensor elements used for the output computation: $\max(0, 1, 3, 4) = 4$.

The output tensor in Fig. 6.5.1 has a height of 2 and a width of 2. The four elements are derived from the maximum value in each pooling window:

$$\begin{aligned}\max(0, 1, 3, 4) &= 4,\\ \max(1, 2, 4, 5) &= 5,\\ \max(3, 4, 6, 7) &= 7,\\ \max(4, 5, 7, 8) &= 8.\end{aligned} \tag{6.5.1}$$

A pooling layer with a pooling window shape of $p \times q$ is called a $p \times q$ pooling layer. The pooling operation is called $p \times q$ pooling.

Let us return to the object edge detection example mentioned at the beginning of this section. Now we will use the output of the convolutional layer as the input for 2×2 maximum pooling. Set the convolutional layer input as `X` and the pooling layer output as `Y`. Whether or not the values of `X[i, j]` and `X[i, j + 1]` are different, or `X[i, j + 1]` and `X[i, j + 2]` are different, the pooling layer always outputs `Y[i, j] = 1`. That is to say, using the 2×2 maximum pooling layer, we can still detect if the pattern recognized by the convolutional layer moves no more than one element in height or width.

In the code below, we implement the forward propagation of the pooling layer in the `pool2d` function. This function is similar to the `corr2d` function in Section 6.2. However, here we have no kernel, computing the output as either the maximum or the average of each region in the input.

```
from mxnet import np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = np.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i:i + p_h, j:j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i:i + p_h, j:j + p_w].mean()
    return Y
```

We can construct the input tensor `X` in Fig. 6.5.1 to validate the output of the two-dimensional maximum pooling layer.

```
X = np.array([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
array([[4., 5.],
       [7., 8.]])
```

Also, we experiment with the average pooling layer.

```
pool2d(X, (2, 2), 'avg')
```

```
array([[2., 3.],
       [5., 6.]])
```

6.5.2 Padding and Stride

As with convolutional layers, pooling layers can also change the output shape. And as before, we can alter the operation to achieve a desired output shape by padding the input and adjusting the stride. We can demonstrate the use of padding and strides in pooling layers via the built-in two-dimensional maximum pooling layer from the deep learning framework. We first construct an input tensor `X` whose shape has four dimensions, where the number of examples (batch size) and number of channels are both 1.

```
X = np.arange(16, dtype=np.float32).reshape((1, 1, 4, 4))
X
```

```
array([[[[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.]]]])
```

By default, the stride and the pooling window in the instance from the framework's built-in class have the same shape. Below, we use a pooling window of shape `(3, 3)`, so we get a stride shape of `(3, 3)` by default.

```
pool2d = nn.MaxPool2D(3)
# Because there are no model parameters in the pooling layer, we do not need
# to call the parameter initialization function
pool2d(X)
```

```
array([[[[10.]]]])
```

The stride and padding can be manually specified.

```
pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)
```

```
array([[[[ 5.,  7.],
         [13., 15.]]]])
```

Of course, we can specify an arbitrary rectangular pooling window and specify the padding and stride for height and width, respectively.

```
pool2d = nn.MaxPool2D((2, 3), padding=(0, 1), strides=(2, 3))
pool2d(X)
```

```
array([[[[ 5.,  7.],
         [13., 15.]]]])
```

6.5.3 Multiple Channels

When processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing the inputs up over channels as in a convolutional layer. This means that the number of output channels for the pooling layer is the same as the number of input channels. Below, we will concatenate tensors `X` and `X + 1` on the channel dimension to construct an input with 2 channels.

```
X = np.concatenate((X, X + 1), 1)
X
```

```
array([[[[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.]],
```

(continues on next page)

(continued from previous page)

```
       [[ 1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.],
        [ 9., 10., 11., 12.],
        [13., 14., 15., 16.]]])
```

As we can see, the number of output channels is still 2 after pooling.

```
pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)
```

```
array([[[[ 5.,  7.],
         [13., 15.]],

        [[ 6.,  8.],
         [14., 16.]]]])
```

Summary

- Taking the input elements in the pooling window, the maximum pooling operation assigns the maximum value as the output and the average pooling operation assigns the average value as the output.
- One of the major benefits of a pooling layer is to alleviate the excessive sensitivity of the convolutional layer to location.
- We can specify the padding and stride for the pooling layer.
- Maximum pooling, combined with a stride larger than 1 can be used to reduce the spatial dimensions (e.g., width and height).
- The pooling layer's number of output channels is the same as the number of input channels.

Exercises

1. Can you implement average pooling as a special case of a convolution layer? If so, do it.
2. Can you implement maximum pooling as a special case of a convolution layer? If so, do it.
3. What is the computational cost of the pooling layer? Assume that the input to the pooling layer is of size $c \times h \times w$, the pooling window has a shape of $p_h \times p_w$ with a padding of (p_h, p_w) and a stride of (s_h, s_w).
4. Why do you expect maximum pooling and average pooling to work differently?
5. Do we need a separate minimum pooling layer? Can you replace it with another operation?
6. Is there another operation between average and maximum pooling that you could consider (hint: recall the softmax)? Why might it not be so popular?

Discussions[89]

[89] https://discuss.d2l.ai/t/71

6.6 Convolutional Neural Networks (LeNet)

We now have all the ingredients required to assemble a fully-functional CNN. In our earlier encounter with image data, we applied a softmax regression model (Section 3.6) and an MLP model (Section 4.2) to pictures of clothing in the Fashion-MNIST dataset. To make such data amenable to softmax regression and MLPs, we first flattened each image from a 28×28 matrix into a fixed-length 784-dimensional vector, and thereafter processed them with fully-connected layers. Now that we have a handle on convolutional layers, we can retain the spatial structure in our images. As an additional benefit of replacing fully-connected layers with convolutional layers, we will enjoy more parsimonious models that require far fewer parameters.

In this section, we will introduce *LeNet*, among the first published CNNs to capture wide attention for its performance on computer vision tasks. The model was introduced by (and named for) Yann LeCun, then a researcher at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images (LeCun et al., 1998). This work represented the culmination of a decade of research developing the technology. In 1989, LeCun published the first study to successfully train CNNs via backpropagation.

At the time LeNet achieved outstanding results matching the performance of support vector machines, then a dominant approach in supervised learning. LeNet was eventually adapted to recognize digits for processing deposits in ATM machines. To this day, some ATMs still run the code that Yann and his colleague Leon Bottou wrote in the 1990s!

6.6.1 LeNet

At a high level, LeNet (LeNet-5) consists of two parts: (i) a convolutional encoder consisting of two convolutional layers; and (ii) a dense block consisting of three fully-connected layers; The architecture is summarized in Fig. 6.6.1.

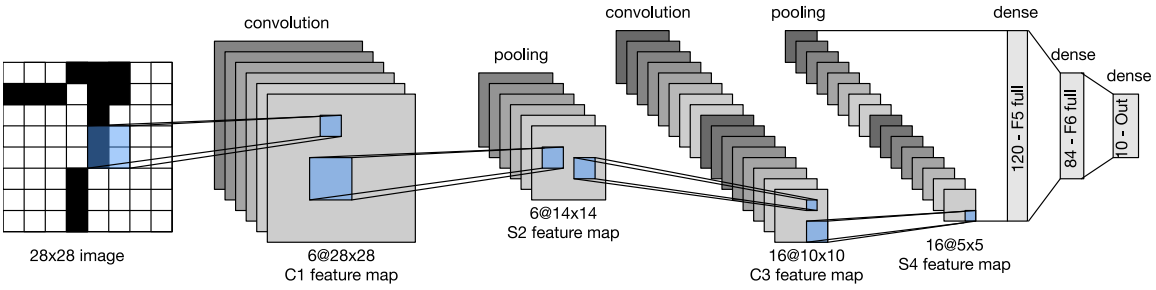


Fig. 6.6.1: Data flow in LeNet. The input is a handwritten digit, the output a probability over 10 possible outcomes.

The basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation. Note that while ReLUs and max-pooling work better, these discoveries had not yet been made in the 1990s. Each convolutional layer uses a 5×5 kernel and a sigmoid activation function. These layers map spatially arranged inputs to a number

of two-dimensional feature maps, typically increasing the number of channels. The first convolutional layer has 6 output channels, while the second has 16. Each 2×2 pooling operation (stride 2) reduces dimensionality by a factor of 4 via spatial downsampling. The convolutional block emits an output with shape given by (batch size, number of channel, height, width).

In order to pass output from the convolutional block to the dense block, we must flatten each example in the minibatch. In other words, we take this four-dimensional input and transform it into the two-dimensional input expected by fully-connected layers: as a reminder, the two-dimensional representation that we desire uses the first dimension to index examples in the minibatch and the second to give the flat vector representation of each example. LeNet's dense block has three fully-connected layers, with 120, 84, and 10 outputs, respectively. Because we are still performing classification, the 10-dimensional output layer corresponds to the number of possible output classes.

While getting to the point where you truly understand what is going on inside LeNet may have taken a bit of work, hopefully the following code snippet will convince you that implementing such models with modern deep learning frameworks is remarkably simple. We need only to instantiate a `Sequential` block and chain together the appropriate layers.

```
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

net = nn.Sequential()
net.add(
    nn.Conv2D(channels=6, kernel_size=5, padding=2, activation='sigmoid'),
    nn.AvgPool2D(pool_size=2, strides=2),
    nn.Conv2D(channels=16, kernel_size=5, activation='sigmoid'),
    nn.AvgPool2D(pool_size=2, strides=2),
    # `Dense` will transform an input of the shape (batch size, number of
    # channels, height, width) into an input of the shape (batch size,
    # number of channels * height * width) automatically by default
    nn.Dense(120, activation='sigmoid'), nn.Dense(84, activation='sigmoid'),
    nn.Dense(10))
```

We took a small liberty with the original model, removing the Gaussian activation in the final layer. Other than that, this network matches the original LeNet-5 architecture.

By passing a single-channel (black and white) 28×28 image through the network and printing the output shape at each layer, we can inspect the model to make sure that its operations line up with what we expect from Fig. 6.6.2.

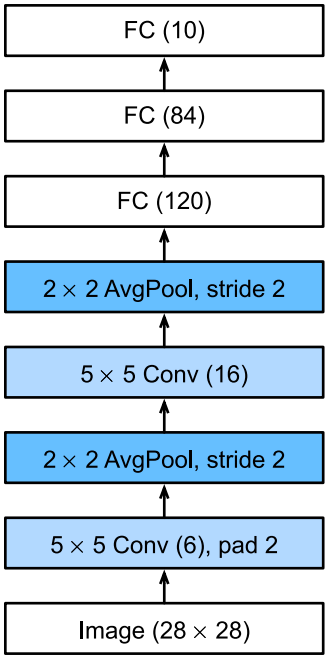


Fig. 6.6.2: Compressed notation for LeNet-5.

```
X = np.random.uniform(size=(1, 1, 28, 28))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

```
conv0 output shape:  (1, 6, 28, 28)
pool0 output shape:  (1, 6, 14, 14)
conv1 output shape:  (1, 16, 10, 10)
pool1 output shape:  (1, 16, 5, 5)
dense0 output shape:         (1, 120)
dense1 output shape:         (1, 84)
dense2 output shape:         (1, 10)
```

Note that the height and width of the representation at each layer throughout the convolutional block is reduced (compared with the previous layer). The first convolutional layer uses 2 pixels of padding to compensate for the reduction in height and width that would otherwise result from using a 5×5 kernel. In contrast, the second convolutional layer forgoes padding, and thus the height and width are both reduced by 4 pixels. As we go up the stack of layers, the number of channels increases layer-over-layer from 1 in the input to 6 after the first convolutional layer and 16 after the second convolutional layer. However, each pooling layer halves the height and width. Finally, each fully-connected layer reduces dimensionality, finally emitting an output whose dimension matches the number of classes.

6.6.2 Training

Now that we have implemented the model, let us run an experiment to see how LeNet fares on Fashion-MNIST.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)
```

While CNNs have fewer parameters, they can still be more expensive to compute than similarly deep MLPs because each parameter participates in many more multiplications. If you have access to a GPU, this might be a good time to put it into action to speed up training.

For evaluation, we need to make a slight modification to the `evaluate_accuracy` function that we described in Section 3.6. Since the full dataset is in the main memory, we need to copy it to the GPU memory before the model uses GPU to compute with the dataset.

```
def evaluate_accuracy_gpu(net, data_iter, device=None):  #@save
    """Compute the accuracy for a model on a dataset using a GPU."""
    if not device:  # Query the first device where the first parameter is on
        device = list(net.collect_params().values())[0].list_ctx()[0]
    # No. of correct predictions, no. of predictions
    metric = d2l.Accumulator(2)
    for X, y in data_iter:
        X, y = X.as_in_ctx(device), y.as_in_ctx(device)
        metric.add(d2l.accuracy(net(X), y), y.size)
    return metric[0] / metric[1]
```

We also need to update our training function to deal with GPUs. Unlike the `train_epoch_ch3` defined in Section 3.6, we now need to move each minibatch of data to our designated device (hopefully, the GPU) prior to making the forward and backward propagations.

The training function `train_ch6` is also similar to `train_ch3` defined in Section 3.6. Since we will be implementing networks with many layers going forward, we will rely primarily on high-level APIs. The following training function assumes a model created from high-level APIs as input and is optimized accordingly. We initialize the model parameters on the device indicated by the `device` argument, using Xavier initialization as introduced in Section 4.8.2. Just as with MLPs, our loss function is cross-entropy, and we minimize it via minibatch stochastic gradient descent. Since each epoch takes tens of seconds to run, we visualize the training loss more frequently.

```
#@save
def train_ch6(net, train_iter, test_iter, num_epochs, lr, device):
    """Train a model with a GPU (defined in Chapter 6)."""
    net.initialize(force_reinit=True, ctx=device, init=init.Xavier())
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        # Sum of training loss, sum of training accuracy, no. of examples
        metric = d2l.Accumulator(3)
        for i, (X, y) in enumerate(train_iter):
            timer.start()
```

(continues on next page)

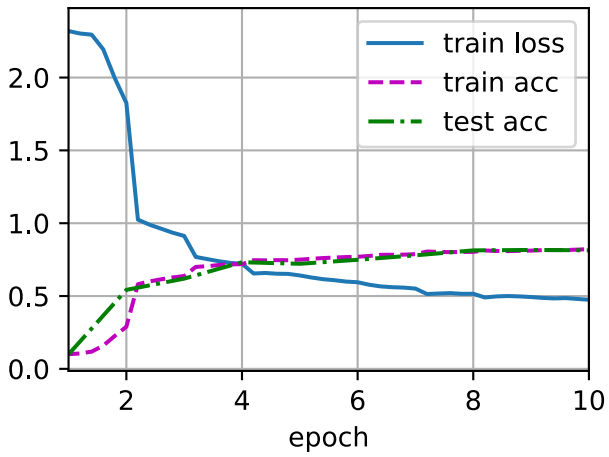
(continued from previous page)

```
            # Here is the major difference from `d2l.train_epoch_ch3`
            X, y = X.as_in_ctx(device), y.as_in_ctx(device)
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y)
            l.backward()
            trainer.step(X.shape[0])
            metric.add(l.sum(), d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (train_l, train_acc, None))
        test_acc = evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(device)}')
```

Now let us train and evaluate the LeNet-5 model.

```
lr, num_epochs = 0.9, 10
train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.474, train acc 0.822, test acc 0.814
45274.3 examples/sec on gpu(0)
```



Summary

- A CNN is a network that employs convolutional layers.
- In a CNN, we interleave convolutions, nonlinearities, and (often) pooling operations.
- In a CNN, convolutional layers are typically arranged so that they gradually decrease the spatial resolution of the representations, while increasing the number of channels.
- In traditional CNNs, the representations encoded by the convolutional blocks are processed by one or more fully-connected layers prior to emitting output.
- LeNet was arguably the first successful deployment of such a network.

Exercises

1. Replace the average pooling with maximum pooling. What happens?
2. Try to construct a more complex network based on LeNet to improve its accuracy.
    1. Adjust the convolution window size.
    2. Adjust the number of output channels.
    3. Adjust the activation function (e.g., ReLU).
    4. Adjust the number of convolution layers.
    5. Adjust the number of fully connected layers.
    6. Adjust the learning rates and other training details (e.g., initialization and number of epochs.)
3. Try out the improved network on the original MNIST dataset.
4. Display the activations of the first and second layer of LeNet for different inputs (e.g., sweaters and coats).

Discussions[90]

[90] https://discuss.d2l.ai/t/73

7 | Modern Convolutional Neural Networks

Now that we understand the basics of wiring together CNNs, we will take you through a tour of modern CNN architectures. In this chapter, each section corresponds to a significant CNN architecture that was at some point (or currently) the base model upon which many research projects and deployed systems were built. Each of these networks was briefly a dominant architecture and many were winners or runners-up in the ImageNet competition, which has served as a barometer of progress on supervised learning in computer vision since 2010.

These models include AlexNet, the first large-scale network deployed to beat conventional computer vision methods on a large-scale vision challenge; the VGG network, which makes use of a number of repeating blocks of elements; the network in network (NiN) which convolves whole neural networks patch-wise over inputs; GoogLeNet, which uses networks with parallel concatenations; residual networks (ResNet), which remain the most popular off-the-shelf architecture in computer vision; and densely connected networks (DenseNet), which are expensive to compute but have set some recent benchmarks.

While the idea of *deep* neural networks is quite simple (stack together a bunch of layers), performance can vary wildly across architectures and hyperparameter choices. The neural networks described in this chapter are the product of intuition, a few mathematical insights, and a whole lot of trial and error. We present these models in chronological order, partly to convey a sense of the history so that you can form your own intuitions about where the field is heading and perhaps develop your own architectures. For instance, batch normalization and residual connections described in this chapter have offered two popular ideas for training and designing deep models.

7.1 Deep Convolutional Neural Networks (AlexNet)

Although CNNs were well known in the computer vision and machine learning communities following the introduction of LeNet, they did not immediately dominate the field. Although LeNet achieved good results on early small datasets, the performance and feasibility of training CNNs on larger, more realistic datasets had yet to be established. In fact, for much of the intervening time between the early 1990s and the watershed results of 2012, neural networks were often surpassed by other machine learning methods, such as support vector machines.

For computer vision, this comparison is perhaps not fair. That is although the inputs to convolutional networks consist of raw or lightly-processed (e.g., by centering) pixel values, practitioners would never feed raw pixels into traditional models. Instead, typical computer vision pipelines consisted of manually engineering feature extraction pipelines. Rather than *learn the features*, the features were *crafted*. Most of the progress came from having more clever ideas for features, and the learning algorithm was often relegated to an afterthought.

Although some neural network accelerators were available in the 1990s, they were not yet sufficiently powerful to make deep multichannel, multilayer CNNs with a large number of parameters. Moreover, datasets were still relatively small. Added to these obstacles, key tricks for training neural networks including parameter initialization heuristics, clever variants of stochastic gradient descent, non-squashing activation functions, and effective regularization techniques were still missing.

Thus, rather than training *end-to-end* (pixel to classification) systems, classical pipelines looked more like this:

1. Obtain an interesting dataset. In early days, these datasets required expensive sensors (at the time, 1 megapixel images were state-of-the-art).
2. Preprocess the dataset with hand-crafted features based on some knowledge of optics, geometry, other analytic tools, and occasionally on the serendipitous discoveries of lucky graduate students.
3. Feed the data through a standard set of feature extractors such as the SIFT (scale-invariant feature transform) (Lowe, 2004), the SURF (speeded up robust features) (Bay et al., 2006), or any number of other hand-tuned pipelines.
4. Dump the resulting representations into your favorite classifier, likely a linear model or kernel method, to train a classifier.

If you spoke to machine learning researchers, they believed that machine learning was both important and beautiful. Elegant theories proved the properties of various classifiers. The field of machine learning was thriving, rigorous, and eminently useful. However, if you spoke to a computer vision researcher, you would hear a very different story. The dirty truth of image recognition, they would tell you, is that features, not learning algorithms, drove progress. Computer vision researchers justifiably believed that a slightly bigger or cleaner dataset or a slightly improved feature-extraction pipeline mattered far more to the final accuracy than any learning algorithm.

7.1.1 Learning Representations

Another way to cast the state of affairs is that the most important part of the pipeline was the representation. And up until 2012 the representation was calculated mechanically. In fact, engineering a new set of feature functions, improving results, and writing up the method was a prominent genre of paper. SIFT (Lowe, 2004), SURF (Bay et al., 2006), HOG (histograms of oriented gradient) (Dalal & Triggs, 2005), bags of visual words[91] and similar feature extractors ruled the roost.

Another group of researchers, including Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari, and Juergen Schmidhuber, had different plans. They believed that features themselves ought to be learned. Moreover, they believed that to be reasonably complex, the features ought to be hierarchically composed with multiple jointly learned layers, each with learnable parameters. In the case of an image, the lowest layers might come to detect edges, colors, and textures. Indeed, Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton proposed a new variant of a CNN, *AlexNet*, that achieved excellent performance in the 2012 ImageNet challenge. AlexNet was named after Alex Krizhevsky, the first author of the breakthrough ImageNet classification paper (Krizhevsky et al., 2012).

Interestingly in the lowest layers of the network, the model learned feature extractors that resembled some traditional filters. Fig. 7.1.1 is reproduced from the AlexNet paper (Krizhevsky et al., 2012) and describes lower-level image descriptors.

[91] https://en.wikipedia.org/wiki/Bag-of-words_model_in_computer_vision

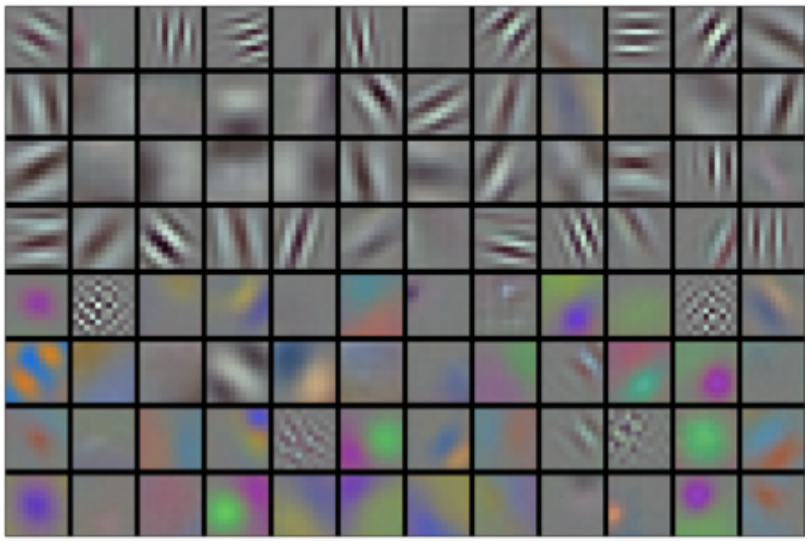


Fig. 7.1.1: Image filters learned by the first layer of AlexNet.

Higher layers in the network might build upon these representations to represent larger structures, like eyes, noses, blades of grass, and so on. Even higher layers might represent whole objects like people, airplanes, dogs, or frisbees. Ultimately, the final hidden state learns a compact representation of the image that summarizes its contents such that data belonging to different categories can be easily separated.

While the ultimate breakthrough for many-layered CNNs came in 2012, a core group of researchers had dedicated themselves to this idea, attempting to learn hierarchical representations of visual data for many years. The ultimate breakthrough in 2012 can be attributed to two key factors.

Missing Ingredient: Data

Deep models with many layers require large amounts of data in order to enter the regime where they significantly outperform traditional methods based on convex optimizations (e.g., linear and kernel methods). However, given the limited storage capacity of computers, the relative expense of sensors, and the comparatively tighter research budgets in the 1990s, most research relied on tiny datasets. Numerous papers addressed the UCI collection of datasets, many of which contained only hundreds or (a few) thousands of images captured in unnatural settings with low resolution.

In 2009, the ImageNet dataset was released, challenging researchers to learn models from 1 million examples, 1000 each from 1000 distinct categories of objects. The researchers, led by Fei-Fei Li, who introduced this dataset leveraged Google Image Search to prefilter large candidate sets for each category and employed the Amazon Mechanical Turk crowdsourcing pipeline to confirm for each image whether it belonged to the associated category. This scale was unprecedented. The associated competition, dubbed the ImageNet Challenge pushed computer vision and machine learning research forward, challenging researchers to identify which models performed best at a greater scale than academics had previously considered.

Deep learning models are voracious consumers of compute cycles. Training can take hundreds of epochs, and each iteration requires passing data through many layers of computationally-expensive linear algebra operations. This is one of the main reasons why in the 1990s and early 2000s, simple algorithms based on the more-efficiently optimized convex objectives were preferred.

Graphical processing units (GPUs) proved to be a game changer in making deep learning feasible. These chips had long been developed for accelerating graphics processing to benefit computer games. In particular, they were optimized for high throughput 4×4 matrix-vector products, which are needed for many computer graphics tasks. Fortunately, this math is strikingly similar to that required to calculate convolutional layers. Around that time, NVIDIA and ATI had begun optimizing GPUs for general computing operations, going as far as to market them as *general-purpose GPUs* (GPGPU).

To provide some intuition, consider the cores of a modern microprocessor (CPU). Each of the cores is fairly powerful running at a high clock frequency and sporting large caches (up to several megabytes of L3). Each core is well-suited to executing a wide range of instructions, with branch predictors, a deep pipeline, and other bells and whistles that enable it to run a large variety of programs. This apparent strength, however, is also its Achilles heel: general-purpose cores are very expensive to build. They require lots of chip area, a sophisticated support structure (memory interfaces, caching logic between cores, high-speed interconnects, and so on), and they are comparatively bad at any single task. Modern laptops have up to 4 cores, and even high-end servers rarely exceed 64 cores, simply because it is not cost effective.

By comparison, GPUs consist of $100 \sim 1000$ small processing elements (the details differ somewhat between NVIDIA, ATI, ARM and other chip vendors), often grouped into larger groups (NVIDIA calls them warps). While each core is relatively weak, sometimes even running at sub-1GHz clock frequency, it is the total number of such cores that makes GPUs orders of magnitude faster than CPUs. For instance, NVIDIA's recent Volta generation offers up to 120 TFlops per chip for specialized instructions (and up to 24 TFlops for more general-purpose ones), while floating point performance of CPUs has not exceeded 1 TFlop to date. The reason for why this is possible is actually quite simple: first, power consumption tends to grow *quadratically* with clock frequency. Hence, for the power budget of a CPU core that runs 4 times faster (a typical number), you can use 16 GPU cores at $1/4$ the speed, which yields $16 \times 1/4 = 4$ times the performance. Furthermore, GPU cores are much simpler (in fact, for a long time they were not even *able* to execute general-purpose code), which makes them more energy efficient. Last, many operations in deep learning require high memory bandwidth. Again, GPUs shine here with buses that are at least 10 times as wide as many CPUs.

Back to 2012. A major breakthrough came when Alex Krizhevsky and Ilya Sutskever implemented a deep CNN that could run on GPU hardware. They realized that the computational bottlenecks in CNNs, convolutions and matrix multiplications, are all operations that could be parallelized in hardware. Using two NVIDIA GTX 580s with 3GB of memory, they implemented fast convolutions. The code cuda-convnet[92] was good enough that for several years it was the industry standard and powered the first couple years of the deep learning boom.

[92] https://code.google.com/archive/p/cuda-convnet/

7.1.2 AlexNet

AlexNet, which employed an 8-layer CNN, won the ImageNet Large Scale Visual Recognition Challenge 2012 by a phenomenally large margin. This network showed, for the first time, that the features obtained by learning can transcend manually-designed features, breaking the previous paradigm in computer vision.

The architectures of AlexNet and LeNet are very similar, as Fig. 7.1.2 illustrates. Note that we provide a slightly streamlined version of AlexNet removing some of the design quirks that were needed in 2012 to make the model fit on two small GPUs.

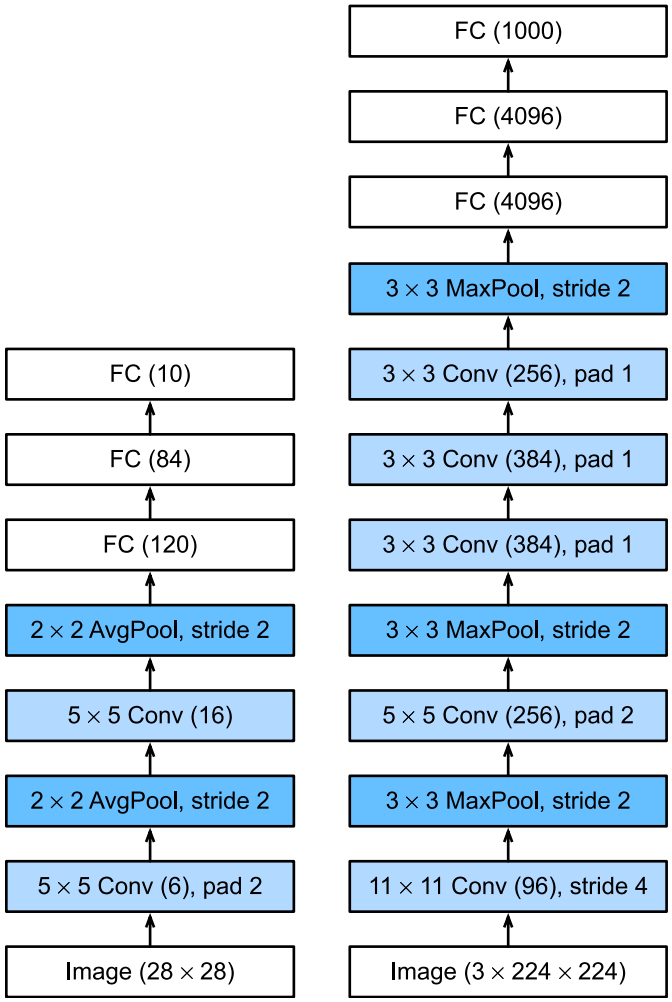


Fig. 7.1.2: From LeNet (left) to AlexNet (right).

The design philosophies of AlexNet and LeNet are very similar, but there are also significant differences. First, AlexNet is much deeper than the comparatively small LeNet5. AlexNet consists of eight layers: five convolutional layers, two fully-connected hidden layers, and one fully-connected output layer. Second, AlexNet used the ReLU instead of the sigmoid as its activation function. Let us delve into the details below.

Architecture

In AlexNet's first layer, the convolution window shape is 11×11. Since most images in ImageNet are more than ten times higher and wider than the MNIST images, objects in ImageNet data tend to occupy more pixels. Consequently, a larger convolution window is needed to capture the object. The convolution window shape in the second layer is reduced to 5×5, followed by 3×3. In addition, after the first, second, and fifth convolutional layers, the network adds maximum pooling layers with a window shape of 3×3 and a stride of 2. Moreover, AlexNet has ten times more convolution channels than LeNet.

After the last convolutional layer there are two fully-connected layers with 4096 outputs. These two huge fully-connected layers produce model parameters of nearly 1 GB. Due to the limited memory in early GPUs, the original AlexNet used a dual data stream design, so that each of their two GPUs could be responsible for storing and computing only its half of the model. Fortunately, GPU memory is comparatively abundant now, so we rarely need to break up models across GPUs these days (our version of the AlexNet model deviates from the original paper in this aspect).

Activation Functions

Besides, AlexNet changed the sigmoid activation function to a simpler ReLU activation function. On one hand, the computation of the ReLU activation function is simpler. For example, it does not have the exponentiation operation found in the sigmoid activation function. On the other hand, the ReLU activation function makes model training easier when using different parameter initialization methods. This is because, when the output of the sigmoid activation function is very close to 0 or 1, the gradient of these regions is almost 0, so that backpropagation cannot continue to update some of the model parameters. In contrast, the gradient of the ReLU activation function in the positive interval is always 1. Therefore, if the model parameters are not properly initialized, the sigmoid function may obtain a gradient of almost 0 in the positive interval, so that the model cannot be effectively trained.

Capacity Control and Preprocessing

AlexNet controls the model complexity of the fully-connected layer by dropout (Section 4.6), while LeNet only uses weight decay. To augment the data even further, the training loop of AlexNet added a great deal of image augmentation, such as flipping, clipping, and color changes. This makes the model more robust and the larger sample size effectively reduces overfitting. We will discuss data augmentation in greater detail in Section 13.1.

```
from mxnet import np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

net = nn.Sequential()
# Here, we use a larger 11 x 11 window to capture objects. At the same time,
# we use a stride of 4 to greatly reduce the height and width of the output.
# Here, the number of output channels is much larger than that in LeNet
net.add(
    nn.Conv2D(96, kernel_size=11, strides=4, activation='relu'),
```

(continues on next page)

(continued from previous page)

```
    nn.MaxPool2D(pool_size=3, strides=2),
    # Make the convolution window smaller, set padding to 2 for consistent
    # height and width across the input and output, and increase the
    # number of output channels
    nn.Conv2D(256, kernel_size=5, padding=2, activation='relu'),
    nn.MaxPool2D(pool_size=3, strides=2),
    # Use three successive convolutional layers and a smaller convolution
    # window. Except for the final convolutional layer, the number of
    # output channels is further increased. Pooling layers are not used to
    # reduce the height and width of input after the first two
    # convolutional layers
    nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
    nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
    nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'),
    nn.MaxPool2D(pool_size=3, strides=2),
    # Here, the number of outputs of the fully-connected layer is several
    # times larger than that in LeNet. Use the dropout layer to mitigate
    # overfitting
    nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
    nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
    # Output layer. Since we are using Fashion-MNIST, the number of
    # classes is 10, instead of 1000 as in the paper
    nn.Dense(10))
```

We construct a single-channel data example with both height and width of 224 to observe the output shape of each layer. It matches the AlexNet architecture in Fig. 7.1.2.

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

```
conv0 output shape:  (1, 96, 54, 54)
pool0 output shape:  (1, 96, 26, 26)
conv1 output shape:  (1, 256, 26, 26)
pool1 output shape:  (1, 256, 12, 12)
conv2 output shape:  (1, 384, 12, 12)
conv3 output shape:  (1, 384, 12, 12)
conv4 output shape:  (1, 256, 12, 12)
pool2 output shape:  (1, 256, 5, 5)
dense0 output shape:         (1, 4096)
dropout0 output shape:       (1, 4096)
dense1 output shape:         (1, 4096)
dropout1 output shape:       (1, 4096)
dense2 output shape:         (1, 10)
```

7.1.3 Reading the Dataset

Although AlexNet is trained on ImageNet in the paper, we use Fashion-MNIST here since training an ImageNet model to convergence could take hours or days even on a modern GPU. One of the problems with applying AlexNet directly on Fashion-MNIST is that its images have lower resolution (28×28 pixels) than ImageNet images. To make things work, we upsample them to 224×224 (generally not a smart practice, but we do it here to be faithful to the AlexNet architecture). We perform this resizing with the `resize` argument in the `d2l.load_data_fashion_mnist` function.

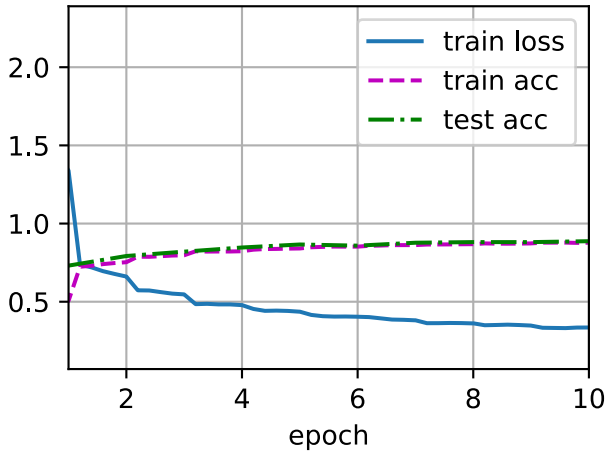
```
batch_size = 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
```

7.1.4 Training

Now, we can start training AlexNet. Compared with LeNet in Section 6.6, the main change here is the use of a smaller learning rate and much slower training due to the deeper and wider network, the higher image resolution, and the more costly convolutions.

```
lr, num_epochs = 0.01, 10
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.335, train acc 0.878, test acc 0.887
3969.6 examples/sec on gpu(0)
```



Summary

- AlexNet has a similar structure to that of LeNet, but uses more convolutional layers and a larger parameter space to fit the large-scale ImageNet dataset.
- Today AlexNet has been surpassed by much more effective architectures but it is a key step from shallow to deep networks that are used nowadays.
- Although it seems that there are only a few more lines in AlexNet's implementation than in LeNet, it took the academic community many years to embrace this conceptual change and

take advantage of its excellent experimental results. This was also due to the lack of efficient computational tools.

- Dropout, ReLU, and preprocessing were the other key steps in achieving excellent performance in computer vision tasks.

Exercises

1. Try increasing the number of epochs. Compared with LeNet, how are the results different? Why?
2. AlexNet may be too complex for the Fashion-MNIST dataset.
    1. Try simplifying the model to make the training faster, while ensuring that the accuracy does not drop significantly.
    2. Design a better model that works directly on 28×28 images.
3. Modify the batch size, and observe the changes in accuracy and GPU memory.
4. Analyze computational performance of AlexNet.
    1. What is the dominant part for the memory footprint of AlexNet?
    2. What is the dominant part for computation in AlexNet?
    3. How about memory bandwidth when computing the results?
5. Apply dropout and ReLU to LeNet-5. Does it improve? How about preprocessing?

Discussions[93]

7.2 Networks Using Blocks (VGG)

While AlexNet offered empirical evidence that deep CNNs can achieve good results, it did not provide a general template to guide subsequent researchers in designing new networks. In the following sections, we will introduce several heuristic concepts commonly used to design deep networks.

Progress in this field mirrors that in chip design where engineers went from placing transistors to logical elements to logic blocks. Similarly, the design of neural network architectures had grown progressively more abstract, with researchers moving from thinking in terms of individual neurons to whole layers, and now to blocks, repeating patterns of layers.

The idea of using blocks first emerged from the Visual Geometry Group[94] (VGG) at Oxford University, in their eponymously-named *VGG* network. It is easy to implement these repeated structures in code with any modern deep learning framework by using loops and subroutines.

[93] https://discuss.d2l.ai/t/75
[94] http://www.robots.ox.ac.uk/~vgg/

7.2.1 VGG Blocks

The basic building block of classic CNNs is a sequence of the following: (i) a convolutional layer with padding to maintain the resolution, (ii) a nonlinearity such as a ReLU, (iii) a pooling layer such as a maximum pooling layer. One VGG block consists of a sequence of convolutional layers, followed by a maximum pooling layer for spatial downsampling. In the original VGG paper (Simonyan & Zisserman, 2014), the authors employed convolutions with 3×3 kernels with padding of 1 (keeping height and width) and 2×2 maximum pooling with stride of 2 (halving the resolution after each block). In the code below, we define a function called `vgg_block` to implement one VGG block.

The function takes two arguments corresponding to the number of convolutional layers `num_convs` and the number of output channels `num_channels`.

```
from mxnet import np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

def vgg_block(num_convs, num_channels):
    blk = nn.Sequential()
    for _ in range(num_convs):
        blk.add(
            nn.Conv2D(num_channels, kernel_size=3, padding=1,
                      activation='relu'))
    blk.add(nn.MaxPool2D(pool_size=2, strides=2))
    return blk
```

7.2.2 VGG Network

Like AlexNet and LeNet, the VGG Network can be partitioned into two parts: the first consisting mostly of convolutional and pooling layers and the second consisting of fully-connected layers. This is depicted in Fig. 7.2.1.

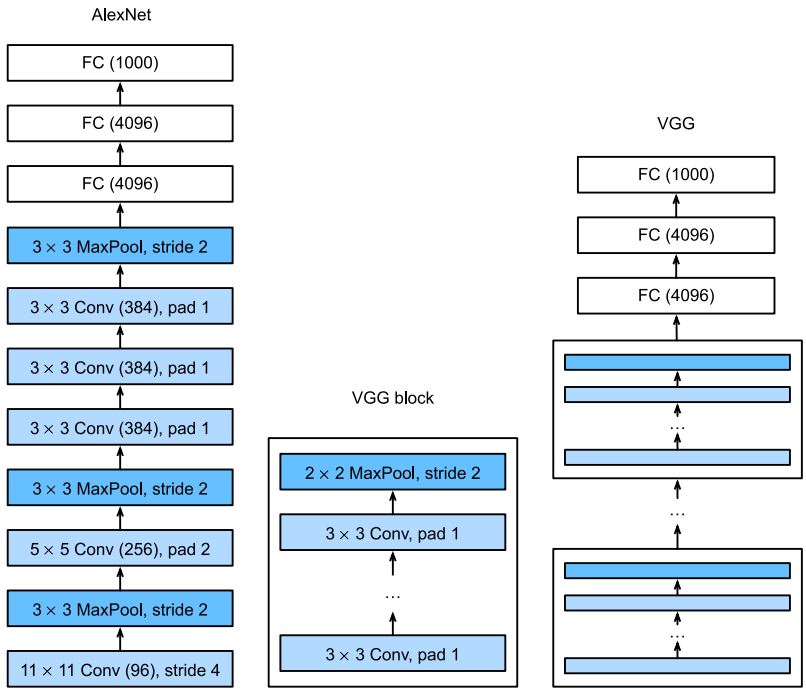


Fig. 7.2.1: From AlexNet to VGG that is designed from building blocks.

The convolutional part of the network connects several VGG blocks from Fig. 7.2.1 (also defined in the `vgg_block` function) in succession. The following variable `conv_arch` consists of a list of tuples (one per block), where each contains two values: the number of convolutional layers and the number of output channels, which are precisely the arguments required to call the `vgg_block` function. The fully-connected part of the VGG network is identical to that covered in AlexNet.

The original VGG network had 5 convolutional blocks, among which the first two have one convolutional layer each and the latter three contain two convolutional layers each. The first block has 64 output channels and each subsequent block doubles the number of output channels, until that number reaches 512. Since this network uses 8 convolutional layers and 3 fully-connected layers, it is often called VGG-11.

```
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

The following code implements VGG-11. This is a simple matter of executing a for-loop over `conv_arch`.

```
def vgg(conv_arch):
    net = nn.Sequential()
    # The convolutional part
    for (num_convs, num_channels) in conv_arch:
        net.add(vgg_block(num_convs, num_channels))
    # The fully-connected part
    net.add(nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
            nn.Dense(4096, activation='relu'), nn.Dropout(0.5), nn.Dense(10))
    return net

net = vgg(conv_arch)
```

Next, we will construct a single-channel data example with a height and width of 224 to observe

the output shape of each layer.

```
net.initialize()
X = np.random.uniform(size=(1, 1, 224, 224))
for blk in net:
    X = blk(X)
    print(blk.name, 'output shape:\t', X.shape)
```

```
sequential1 output shape:    (1, 64, 112, 112)
sequential2 output shape:    (1, 128, 56, 56)
sequential3 output shape:    (1, 256, 28, 28)
sequential4 output shape:    (1, 512, 14, 14)
sequential5 output shape:    (1, 512, 7, 7)
dense0 output shape:         (1, 4096)
dropout0 output shape:       (1, 4096)
dense1 output shape:         (1, 4096)
dropout1 output shape:       (1, 4096)
dense2 output shape:         (1, 10)
```

As you can see, we halve height and width at each block, finally reaching a height and width of 7 before flattening the representations for processing by the fully-connected part of the network.

7.2.3 Training

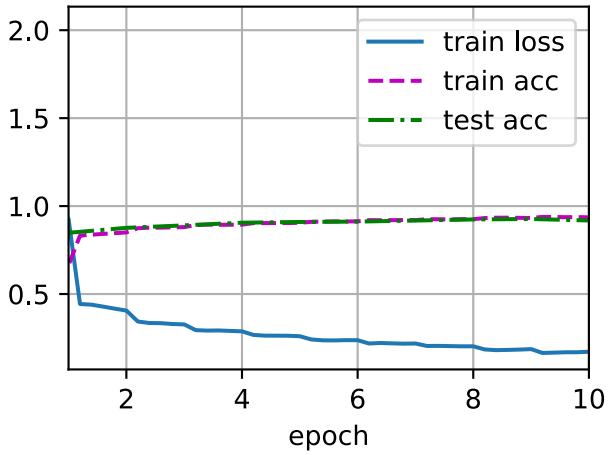
Since VGG-11 is more computationally-heavy than AlexNet we construct a network with a smaller number of channels. This is more than sufficient for training on Fashion-MNIST.

```
ratio = 4
small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
net = vgg(small_conv_arch)
```

Apart from using a slightly larger learning rate, the model training process is similar to that of AlexNet in Section 7.1.

```
lr, num_epochs, batch_size = 0.05, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.172, train acc 0.936, test acc 0.918
1774.2 examples/sec on gpu(0)
```



Summary

- VGG-11 constructs a network using reusable convolutional blocks. Different VGG models can be defined by the differences in the number of convolutional layers and output channels in each block.
- The use of blocks leads to very compact representations of the network definition. It allows for efficient design of complex networks.
- In their VGG paper, Simonyan and Ziserman experimented with various architectures. In particular, they found that several layers of deep and narrow convolutions (i.e., 3×3) were more effective than fewer layers of wider convolutions.

Exercises

1. When printing out the dimensions of the layers we only saw 8 results rather than 11. Where did the remaining 3 layer information go?
2. Compared with AlexNet, VGG is much slower in terms of computation, and it also needs more GPU memory. Analyze the reasons for this.
3. Try changing the height and width of the images in Fashion-MNIST from 224 to 96. What influence does this have on the experiments?
4. Refer to Table 1 in the VGG paper (Simonyan & Zisserman, 2014) to construct other common models, such as VGG-16 or VGG-19.

Discussions[95]

[95] https://discuss.d2l.ai/t/77

7.3 Network in Network (NiN)

LeNet, AlexNet, and VGG all share a common design pattern: extract features exploiting *spatial* structure via a sequence of convolution and pooling layers and then post-process the representations via fully-connected layers. The improvements upon LeNet by AlexNet and VGG mainly lie in how these later networks widen and deepen these two modules. Alternatively, one could imagine using fully-connected layers earlier in the process. However, a careless use of dense layers might give up the spatial structure of the representation entirely, *network in network* (*NiN*) blocks offer an alternative. They were proposed based on a very simple insight: to use an MLP on the channels for each pixel separately (Lin et al., 2013).

7.3.1 NiN Blocks

Recall that the inputs and outputs of convolutional layers consist of four-dimensional tensors with axes corresponding to the example, channel, height, and width. Also recall that the inputs and outputs of fully-connected layers are typically two-dimensional tensors corresponding to the example and feature. The idea behind NiN is to apply a fully-connected layer at each pixel location (for each height and width). If we tie the weights across each spatial location, we could think of this as a 1×1 convolutional layer (as described in Section 6.4) or as a fully-connected layer acting independently on each pixel location. Another way to view this is to think of each element in the spatial dimension (height and width) as equivalent to an example and a channel as equivalent to a feature.

Fig. 7.3.1 illustrates the main structural differences between VGG and NiN, and their blocks. The NiN block consists of one convolutional layer followed by two 1×1 convolutional layers that act as per-pixel fully-connected layers with ReLU activations. The convolution window shape of the first layer is typically set by the user. The subsequent window shapes are fixed to 1×1.

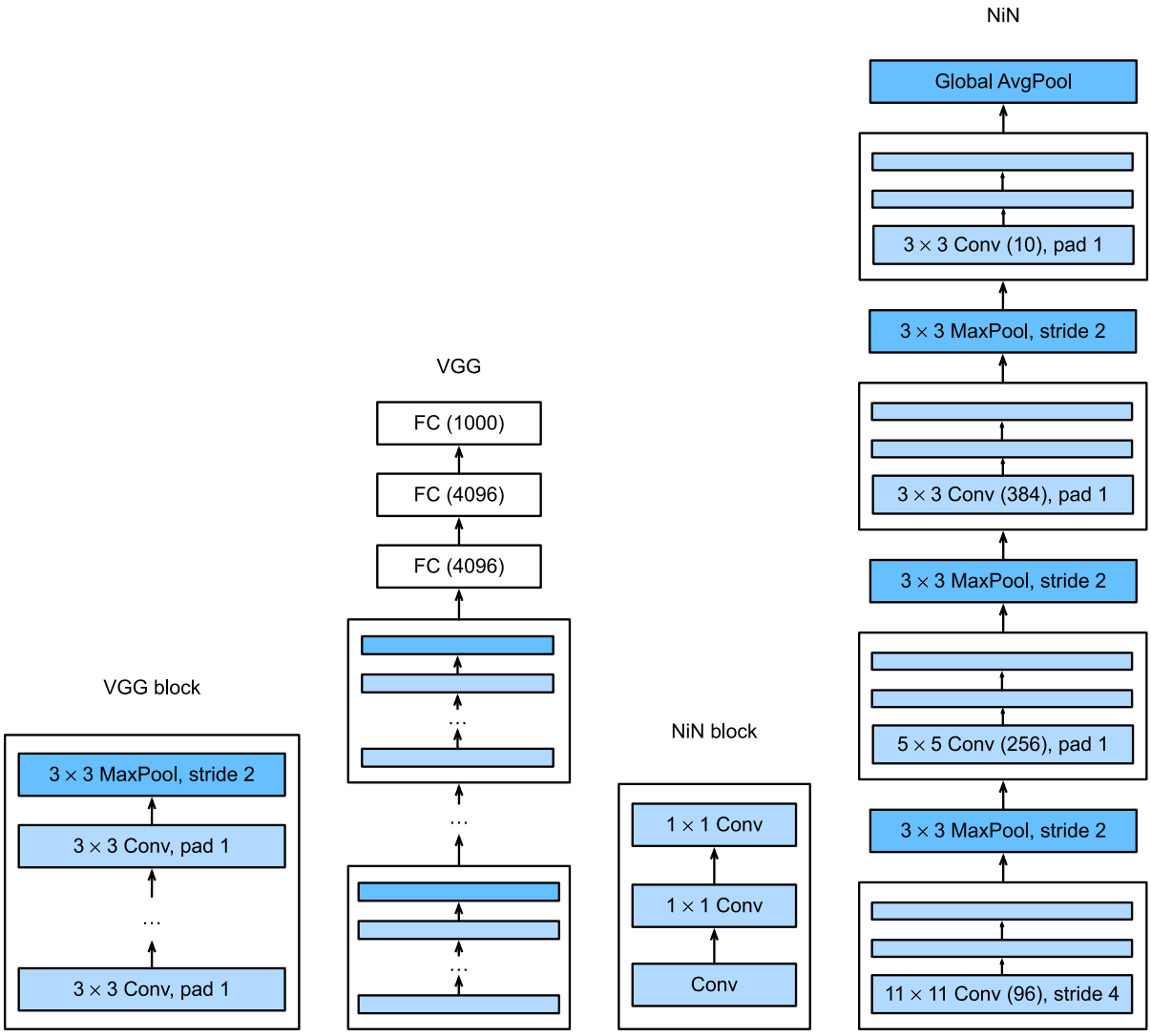


Fig. 7.3.1: Comparing architectures of VGG and NiN, and their blocks.

```
from mxnet import np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

def nin_block(num_channels, kernel_size, strides, padding):
    blk = nn.Sequential()
    blk.add(
        nn.Conv2D(num_channels, kernel_size, strides, padding,
                  activation='relu'),
        nn.Conv2D(num_channels, kernel_size=1, activation='relu'),
        nn.Conv2D(num_channels, kernel_size=1, activation='relu'))
    return blk
```

7.3.2 NiN Model

The original NiN network was proposed shortly after AlexNet and clearly draws some inspiration. NiN uses convolutional layers with window shapes of 11×11, 5×5, and 3×3, and the corresponding numbers of output channels are the same as in AlexNet. Each NiN block is followed by a maximum pooling layer with a stride of 2 and a window shape of 3×3.

One significant difference between NiN and AlexNet is that NiN avoids fully-connected layers altogether. Instead, NiN uses an NiN block with a number of output channels equal to the number of label classes, followed by a *global* average pooling layer, yielding a vector of logits. One advantage of NiN's design is that it significantly reduces the number of required model parameters. However, in practice, this design sometimes requires increased model training time.

```
net = nn.Sequential()
net.add(
    nin_block(96, kernel_size=11, strides=4, padding=0),
    nn.MaxPool2D(pool_size=3, strides=2),
    nin_block(256, kernel_size=5, strides=1, padding=2),
    nn.MaxPool2D(pool_size=3, strides=2),
    nin_block(384, kernel_size=3, strides=1, padding=1),
    nn.MaxPool2D(pool_size=3, strides=2), nn.Dropout(0.5),
    # There are 10 label classes
    nin_block(10, kernel_size=3, strides=1, padding=1),
    # The global average pooling layer automatically sets the window shape
    # to the height and width of the input
    nn.GlobalAvgPool2D(),
    # Transform the four-dimensional output into two-dimensional output
    # with a shape of (batch size, 10)
    nn.Flatten())
```

We create a data example to see the output shape of each block.

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

```
sequential1 output shape:    (1, 96, 54, 54)
pool0 output shape:  (1, 96, 26, 26)
sequential2 output shape:    (1, 256, 26, 26)
pool1 output shape:  (1, 256, 12, 12)
sequential3 output shape:    (1, 384, 12, 12)
pool2 output shape:  (1, 384, 5, 5)
dropout0 output shape:       (1, 384, 5, 5)
sequential4 output shape:    (1, 10, 5, 5)
pool3 output shape:  (1, 10, 1, 1)
flatten0 output shape:       (1, 10)
```