Dive into Deep Learning

Release 0.7

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

CONTENTS

CHAPTER

ONE

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 the youngest among us (the authors) 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 forward-looking academic discipline with a narrow set of real-world applications. And those applications, e.g., speech recognition and computer vision, required so much domain knowledge that they were often regarded as separate areas entirely for which machine learning was one small component. Neural networks then, the antecedents of the deep learning models that we focus on in this book, were regarded as outmoded tools.

In just the past five years, deep learning has taken the world by surprise, driving rapid progress in fields as diverse as computer vision, natural language processing, automatic speech recognition, reinforcement learning, and statistical modeling. 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. This book represents our attempt to make deep learning approachable, teaching you both the *concepts*, the *context*, and the *code*.

1.1 About This Book

1.1.1 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 entrepreneurs 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 mathematics of a given modeling approach; (iii) the optimization algorithms for fitting the models to data; and (iv) and 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.

We started this book project in July 2017 when we needed to explain MXNet's (then new) Gluon interface to our users. At the time, 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 engagine blog posts published on the website Distill[1], or personal blogs, they only covered selected topics in deep learning, and often lacked associated code. On the other hand, while several textbooks have emerged, most notably (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 (1) be freely available for everyone; (2) offer sufficient technical depth to provide a starting point on the path to actually becoming an applied machine learning scientist; (3) include runnable code, showing readers *how* to solve problems in practice; (4) that allowed for rapid updates, both by us and also by the community at large; and (5) be complemented by a forum[2] 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 19.4. We settled on Github to share the source and to allow for edits, 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.

1.1.2 Learning by Doing

Many textbooks teach a series of topics, 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 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 *1 working example, 1 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.

[1] http://distill.pub
[2] http://discuss.mxnet.io

Throughout, we will be working with the MXNet library, which has the rare property of being flexible enough for research while being fast enough for production. 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 `Gluon`'s 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 NDArray and automatic differentiation, and another, more practical example, where we write succinct code using `Gluon`. Once we have taught you how some component works, we can just use the `Gluon` version in subsequent tutorials.

1.1.3 Content and Structure

The book can be roughly divided into three parts, which are presented by different colors in Fig. 1.1.1:

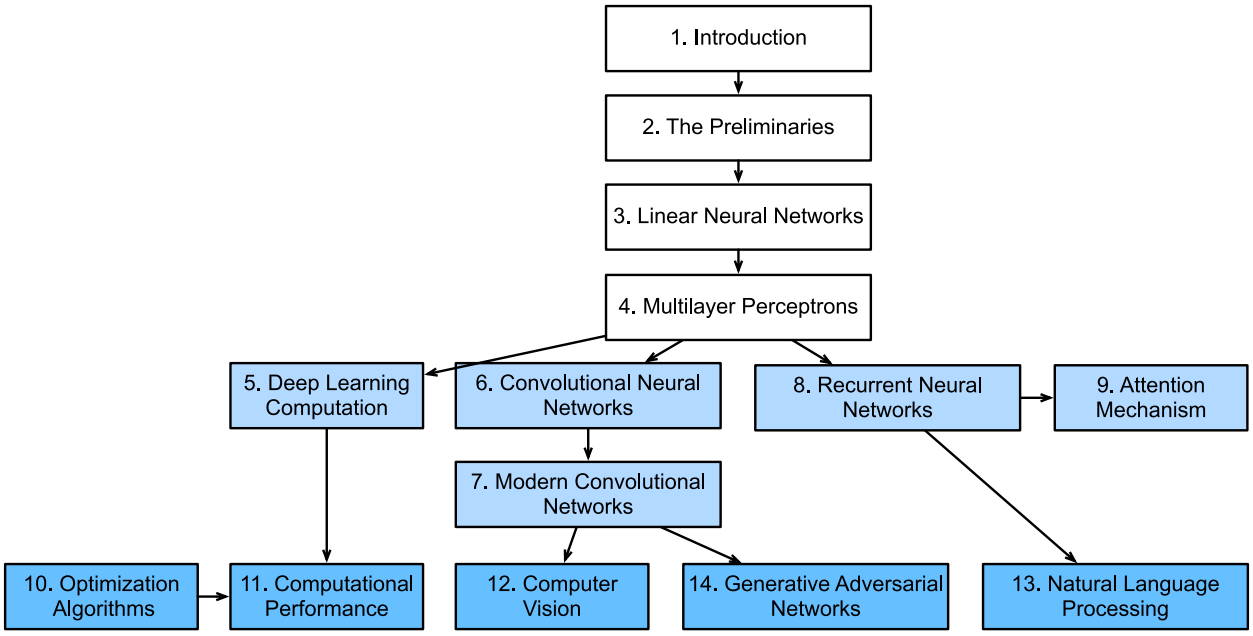


Fig. 1.1.1: Book structure

- The first part covers prerequisites and basics. The first chapter offers an introduction to deep learning Section 3. Then, in Section 4, 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. Section 5 and Section 6 cover the most basic concepts and techniques of deep learning, such as linear regression, multi-layer perceptrons and regularization.
- The next four chapters focus on modern deep learning techniques. Section 7 describes the various key components of deep learning calculations and lays the groundwork for us to subsequently implement more complex models. Next, in Section 8 and Section 9, we introduce Convolutional Neural Networks (CNNs), powerful tools that form the backbone of most modern computer vision systems. Subsequently, in Section 10, we introduce Recurrent Neural Networks (RNNS), models that exploit temporal or sequential structure in data, and are commonly used for natural language processing and time series prediction. In Section 11, we introduce a new class of models that employ a technique called an attention mechanism and that have recently begun to displace RNNs in NLP. These sections will get you up to speed on the basic tools behind most modern applications of deep learning.
- Part three discusses scalability, efficiency and applications. First, in Section 12, we discuss several common optimization algorithms used to train deep learning models. The next chapter, Section 13 examines several key factors that influence the computational performance of your deep learning code. In Section 14 and Section 15, we illus-

trate major applications of deep learning in computer vision and natural language processing, respectively. Finally, presents an emerging family of models called Generative Adversarial Networks (GANs).

1.1.4 Code

Most sections of this book feature executable code because of our belief in the importance of an interactive learning experience in deep learning. At present, certain intuitions can only be developed through 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. Unfortunately, at present, such elegant theories elude us. Despite our best attempts, formal explanations for various techniques are still lacking, both because the mathematics to charactize 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 will be able to provide insights in places the present edition cannot.

Most of the code in this book is based on Apache MXNet. MXNet is an open-source framework for deep learning and 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. However, we plan to keep the online version remain up-to-date. In case you encounter any such problems, please consult *Installation* (page 7) to update your code and runtime environment.

At times, to avoid unnecessary repetition, we encapsulate the frequently-imported and referred-to functions, classes, etc. in this book in the `d2l` package. For any block block such as a function, a class, or multiple imports to be saved in the package, we will mark it with `# Saved in the d2l package for later use`. The `d2l` package is light-weight and only requires the following packages and modules as dependencies:

```
# Saved in the d2l package for later use
from IPython import display
import collections
from collections import defaultdict
import os
import sys
import math
from matplotlib import pyplot as plt
from mxnet import np, npx, autograd, gluon, init, context, image
from mxnet.gluon import nn, rnn
from mxnet.gluon.loss import Loss
from mxnet.gluon.data import Dataset
import random
import re
import time
import tarfile
import zipfile
import pandas as pd
```

We offer a detailed overview of these functions and classes in Section 19.5.

1.1.5 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 (the very basics of) linear algebra, calculus, probability, and Python programming. Moreover, in the Appendix, we provide a refresher on most of the mathematics

covered in this book. Most of the time, we will prioritize intuition and ideas over mathematical rigor. There are many terrific books which can lead the interested reader further. For instance, Linear Analysis by Bela Bollobas (Bollobas, 1999) covers linear algebra and functional analysis in great depth. All of Statistics (Wasserman, 2013) is a terrific guide to statistics. And if you have not used Python before, you may want to peruse this Python tutorial[3].

1.1.6 Forum

Associated with this book, we have launched a discussion forum, located at discuss.mxnet.io[4]. When you have questions on any section of the book, you can find the associated discussion page by scanning the QR code at the end of the section to participate in its discussions. The authors of this book and broader MXNet developer community frequently participate in forum discussions.

1.2 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, vishwesh5, YaYaB, Yuhong Chen, Evgeniy Smirnov, lgov, Simon Corston-Oliver, IgorDzreyev, Ha Nguyen, pmuens, alukovenko, senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, Bowen Li, Aarush Ahuja, prasanth5reddy, 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, ruslo, Rafael Schlatter, liusy182, Giannis Pappas, ruslo, ati-ozgur, qbaza, dchoi77, Adam Gerson. Notably, Brent Werness (Amazon) and Rachel Hu (Amazon) co-authored the *Mathematics for Deep Learning* chapter in the Appendix with us and are the major contributors to that chapter.

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.

1.3 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.mxnet.io/.
- Apache MXNet is a powerful library for coding up deep learning models and running them in parallel across GPU cores.
- Gluon is a high level library that makes it easy to code up deep learning models using Apache MXNet.
- Conda is a Python package manager that ensures that all software dependencies are met.

[3] http://learnpython.org/
[4] https://discuss.mxnet.io/

- All notebooks are available for download on GitHub and the conda configurations needed to run this book's code are expressed in the `environment.yml` file.
- If you plan to run this code on GPUs, do not forget to install the necessary drivers and update your configuration.

1.4 Exercises

1. Register an account on the discussion forum of this book discuss.mxnet.io[5].
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.
4. Create an account on the forum and introduce yourself.

1.5 Scan the QR Code to Discuss[6]

[5] https://discuss.mxnet.io/
[6] https://discuss.mxnet.io/t/2311

CHAPTER

TWO

INSTALLATION

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

2.1 Installing Miniconda

The simplest way to get going will be to install Miniconda[7]. Download the corresponding Miniconda "sh" file from the website and then execute the installation from the command line using `sudo sh <FILENAME>` as follows:

```
# For Mac users (the file name is subject to changes)
sudo sh Miniconda3-latest-MacOSX-x86_64.sh

# For Linux users (the file name is subject to changes)
sudo sh Miniconda3-latest-Linux-x86_64.sh
```

You will be prompted to answer the following questions:

```
Do you accept the license terms? [yes|no]
[no] >>> yes

Miniconda3 will now be installed into this location:
/home/rlhu/miniconda3
  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below
>>> <ENTER>

Do you wish the installer to initialize Miniconda3
by running conda init? [yes|no]
[no] >>> yes
```

After installing miniconda, run the appropriate command (depending on your operating system) to activate conda.

```
# For Mac user
source ~/.bash_profile

# For Linux user
source ~/.bashrc
```

Then create the conda "d2l"" environment and enter `y` for the following inquiries as shown in Fig. 2.1.1.

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

```
conda create --name d2l
```

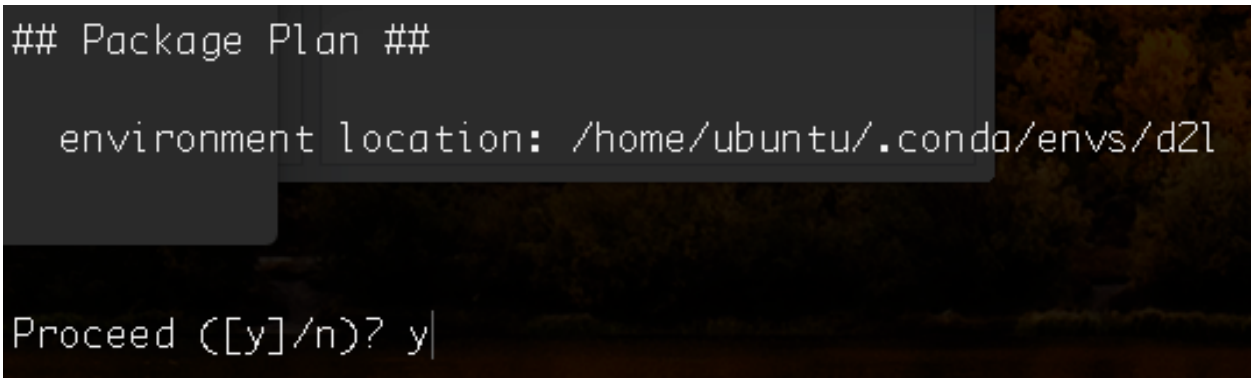


Fig. 2.1.1: Conda create environment d2l.

2.2 Downloading the d2l Notebooks

Next, we need to download the code for this book.

```
sudo apt-get install unzip
mkdir d2l-en && cd d2l-en
wget http://numpy.d2l.ai/d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

Now we will now want to activate the "d2l" environment and install `pip`. Enter `y` for the queries that follow this command.

```
conda activate d2l
conda install python=3.7 pip
```

Finally, install the "d2l" package within the environment "d2l" that we created.

```
pip install git+https://github.com/d2l-ai/d2l-en@numpy2
```

If everything went well up to now then you are almost there. If by some misfortune, something went wrong along the way, please check the following:

1. That you are using `pip` for Python 3 instead of Python 2 by checking `pip --version`. If it is Python 2, then you may check if there is a `pip3` available.
2. That you are using a recent `pip`, such as version 19. If not, you can upgrade it via `pip install --upgrade pip`.
3. Whether you have permission to install system-wide packages. If not, you can install to your home directory by adding the flag `--user` to the pip command, e.g. `pip install d2l --user`.

2.3 Installing MXNet

Before installing `mxnet`, 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 9) for instructions to install a GPU-supported `mxnet`.

Otherwise, you can install the CPU version. 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.

```
# For Windows users
pip install mxnet==1.6.0b20190926

# For Linux and macOS users
pip install mxnet==1.6.0b20190915
```

Once both packages 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. Once in the notebook server, we can run the code for each section of the book.

2.4 Upgrade to a New Version

Both this book and MXNet are keeping improving. Please check a new version from time to time.

1. The URL http://numpy.d2l.ai/d2l-en.zip always points to the latest contents.
2. Please upgrade "d2l" by `pip install git+https://github.com/d2l-ai/d2l-en@numpy2`.
3. For the CPU version, MXNet can be upgraded by `pip uninstall mxnet` then re-running the aforementioned `pip install mxnet==...` command.

2.5 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[8], then you should install a GPU-enabled MXNet. 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 you have installed CUDA 10.1, then you can install the according MXNet version with the following (OS-specific) command:

```
# For Windows users
pip install mxnet-cu101==1.6.0b20190926

# For Linux and macOS users
pip install mxnet-cu101==1.6.0b20190915
```

You may change the last digits according to your CUDA version, e.g., `cu100` for CUDA 10.0 and `cu90` for CUDA 9.0. You can find all available MXNet versions via `pip search mxnet`.

For installation of MXNet on other platforms, please refer to http://numpy.mxnet.io/#installation.

[8] https://developer.nvidia.com/cuda-downloads

2.6 Exercises

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

2.7 Scan the QR Code to Discuss[9]

[9] https://discuss.mxnet.io/t/2315

CHAPTER

THREE

INTRODUCTION

Until recently, nearly every computer program that interact with daily were 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'd 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 ML 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 (ML) is the study of powerful techniques that can *learn* from *experience*. As ML algorithm accumulates more experience, typically in the form of observational data or interactions with an environment, their 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.

3.1 A Motivating Example

Before we could begin 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 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* like 'Alexa', 'Okay, Google' or 'Siri'. Try coding it up in a room by yourself with nothing but a computer and a code editor. How would you write such a program from first principles? Think about it… the problem is hard. Every second, the microphone will collect roughly 44,000 samples. Each sample is a measurement of the amplitude of the sound wave. What rule could map reliably from a snippet of raw audio to confident predictions `{yes, 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 ML.

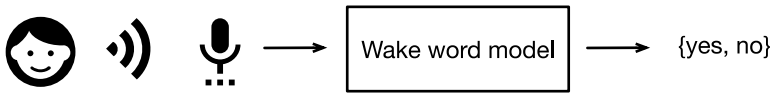


Fig. 3.1.1: Identify an awake word.

Here's 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 the word 'Alexa'. 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 ML 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 it generates a selection among `{yes, no}` as *output*. If all goes according to plan the model's guesses will typically be correct as to whether (or not) the snippet contains the wake word.

If we choose the right family of models, then 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 not likely that our model will recognize 'Alexa', 'Apricot', or any other English word. In deep learning, the *learning* is the process by which we discover the right setting of the knobs

coercing the desired behavior from our model.

The training process usually looks like this:

1. Start off with a randomly initialized model that cannot do anything useful.
2. Grab some of your labeled data (e.g., audio snippets and corresponding `{yes,no}` labels)
3. Tweak the knobs so the model sucks less with respect to those examples
4. Repeat until the model is awesome.

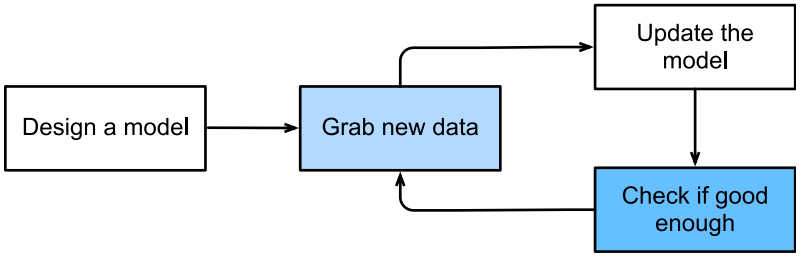


Fig. 3.1.2: A typical training process.

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*. We can "program" a cat detector by providing our machine learning system with many examples of cats and dogs, such as the images below:

			
cat	cat	dog	dog

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 ML can do.

Deep learning is just one among many popular methods for solving machine learning problems. Thus far, we have only talked about machine learning broadly and not deep learning. To see why deep learning is important, we should pause for a moment to highlight a couple crucial points.

First, the problems that we have discussed thus far—learning from raw audio signal, the raw pixel values of images, or mapping between sentences of arbitrary lengths and their counterparts in foreign languages—are problems where deep learning excels and where traditional ML methods faltered. Deep models are *deep* in precisely the sense that they learn many *layers* of computation. It turns out that these many-layered (or hierarchical) models are capable of addressing low-level perceptual data in a way that previous tools could not. In bygone days, the crucial part of applying ML to these problems consisted of coming up with manually-engineered ways of transforming the data into some form amenable to *shallow* models. 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. Secondly, 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.

3.2 The Key Components: Data, Models, and Algorithms

In our *wake-word* example, we described a dataset consisting of audio snippets and binary labels 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* given known *inputs*, given a dataset consisting of examples, for which the labels are known is called *supervised learning*, and it is just one among many *kinds* of machine learning problems. In the next section, we will take a deep dive into the different ML problems. First, we'd like to shed more light on some core components that will follow us around, no matter what kind of ML problem we take on:

1. The *data* that we can learn from.
2. A *model* of how to transform the data.
3. A *loss* function that quantifies the *badness* of our model.
4. An *algorithm* to adjust the model's parameters to minimize the loss.

3.2.1 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* (also called *data points*, *samples*, or *instances*). In order to work with data usefully, we typically need to come up with a suitable numerical representation. Each *example* typically consists of a collection of numerical attributes called *features*. In the supervised learning problems above, a special feature is designated as the prediction *target*, (sometimes called the *label* or *dependent variable*). The given features from which the model must make its predictions can then simply be called the *features*, (or often, the *inputs*, *covariates*, or *independent variables*).

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 a more 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, diagnoses, etc.

When every example is characterized by the same number of numerical values, we say that the data consists 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 means we have one less thing to worry about.

However, not all data can easily be represented as fixed length vectors. While we might expect microscrope 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 resists fixed-length representations even more stubbornly. Consider the customer reviews left on e-commerce sites like Amazon, IMDB, or 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 either do not work without large datasets. Some others work in the low-data regime, but no better than traditional approaches.

Finally it is not enough to have lots of data and to process it cleverly. We need the *right* data. If the data is full of mistakes, or if the chosen features are not predictive of the target quantity of interest, learning is going to fail. The situation is captured well by the cliché: *garbage in, garbage out*. Moreover, poor predictive performance is not the only potential consequence. In sensitive applications of machine learning, like predictive policing, resumé 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 does not merely under-represent some groups, but reflects societal prejudices. For example if past hiring decisions are used to train a predictive model that will be used to screen resumes, then machine learning models could inadvertently capture and automate historical injustices. Note that this can all happen without the data scientist actively conspiring, or even being aware.

3.2.2 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 neural networks, we will discuss some more traditional methods.

3.2.3 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 f for which higher is better, and turn it into a new function f' that is qualitatively identical but for which lower is better by setting $f' = -f$. Because lower is better, these functions are sometimes called *loss functions* or *cost functions*.

When trying to predict numerical values, the most common objective function is squared error $(y-\hat{y})^2$. For classification, the most common objective is to minimize error rate, i.e., the fraction of instances on which our predictions disagree with the ground truth. Some objectives (like squared error) are easy to optimize. Others (like 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. The best values of our model's parameters are learned by minimizing the loss incurred on a *training 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) test data. So we will typically want to split the available data into two partitions: the training data (for fitting model parameters) and the test data (which is held out for evaluation), reporting the following two quantities:

- **Training Error:** The error on that data on which the model was trained. You could think of this as being like a student's scores on practice exams used to prepare for some real exam. Even if the results are encouraging, that does not guarantee success on the final exam.

- **Test Error:** This is the error incurred on an unseen test set. This can deviate significantly from the training error. When a model performs well on the training data 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.

3.2.4 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. The most popular optimization algorithms for neural networks follow an approach called gradient descent. In short, at each step, they check to see, for each parameter, which way the training set loss would move if you perturbed that parameter just a small amount. They then update the parameter in the direction that reduces the loss.

3.3 Kinds of Machine Learning

In the following sections, we discuss a few *kinds* of machine learning problems in greater detail. We begin with a list of *objectives*, i.e., a list of things that we would like machine learning to do. Note that the objectives are complemented with a set of techniques of *how* to accomplish them, including types of data, models, training techniques, etc. The list below is just a sampling of the problems ML can tackle to motivate the reader and provide us with some common language for when we talk about more problems throughout the book.

3.3.1 Supervised learning

Supervised learning addresses the task of predicting *targets* given *inputs*. The targets, which we often call *labels*, are generally denoted by y. The input data, also called the *features* or covariates, are typically denoted $\mathbf{x}$. Each (input, target) pair is called an *examples* or an *instances*. Some times, when the context is clear, we may use the term examples, to refer to a collection of inputs, even when the corresponding targets are unknown. We denote any particular instance with a subscript, typicaly i, for instance ($\mathbf{x}_i$, y_i). A dataset is a collection of n instances $\{\mathbf{x}_i, y_i\}_{i=1}^n$. Our goal is to produce a model f_θ that maps any input $\mathbf{x}_i$ to a prediction $f_\theta(\mathbf{x}_i)$

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 y. The input data $\mathbf{x}$ might be vital signs such as heart rate, diastolic and systolic blood pressure, etc.

The supervision comes into play because for choosing the parameters θ, we (the supervisors) provide the model with a dataset consisting of *labeled examples* ($\mathbf{x}_i, y_i$), where each example $\mathbf{x}_i$ is matched with the correct label.

In probabilistic terms, we typically are interested in estimating the conditional probability $P(y|x)$. 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 CT 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 "predict targets from inputs" 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 (like strings of text or time series data) and for processing fixed-length vector representations. We will visit many of these problems in depth throughout the first 9 parts of this book.

Informally, the learning process looks something like this: Grab a big collection of examples for which the covariates are known and select from them a random subset, acquiring the ground truth labels for each. Sometimes these labels might be available data that has 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.

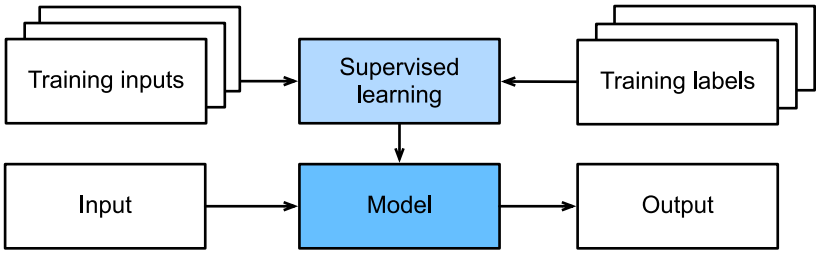


Fig. 3.3.1: Supervised learning.

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: $[100, 0, .5, 60]$. 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. We will continue to denote the feature vector correspond to any example i as $\mathbf{x}_i$ and we can compactly refer to the full table containing all of the feature vectors as X.

What makes a problem a *regression* is actually the outputs. 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 these. The target value, the price of sale, is a *real number*. If you remember the formal definition of the reals you might be scratching your head now. Homes probably never sell for fractions of a cent, let alone prices expressed as irrational numbers. In cases like this, when the target is actually discrete, but where the rounding takes place on a sufficiently fine scale, we will abuse language just a bit cn continue to describe our outputs and targets as real-valued numbers.

We denote any individual target y_i (corresponding to example $\mathbf{x_i}$) and the set of all targets $\mathbf{y}$ (corresponding to all examples X). When our targets take on arbitrary values in some range, we call this a regression problem. Our goal is to produce a model whose predictions closely approximate the actual target values. We denote the predicted target for any instance $\hat{y}_i$. Do not worry if the notation is bogging you down. We will unpack it more thoroughly in the subsequent chapters.

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 Netflix prize[10]. 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.

- 'How many hours will this surgery take?' - *regression*
- 'How many dogs are in this photo?' - *regression*.

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

However, if you can easily pose your problem as 'Is this a _ ?', then it is likely, classification, a different kind of supervised problem that we will cover next. 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 $x_1 = 3$ hours removing gunk from your sewage pipes. Then she sent you a bill of $y_1 = \$350$. Now imagine that your friend hired the same contractor for $x_2 = 2$ hours and that she received a bill of $y_2 = \$250$. 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 points, you could already identify the contractor's pricing structure: \$100 per hour plus \$50 to show up at your house. If you followed that much then you already understand the high-level idea behind linear regression (and you just implicitly designed a linear model with a bias term).

In this case, we could produce the parameters that exactly matched the contractor's prices. Sometimes that is not possible, e.g., if some of the variance owes to some 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 one of two very common losses, the L1 loss[11] where

$$l(y, y') = \sum_i |y_i - y'_i| \tag{3.3.1}$$

and the least mean squares loss, or L2 loss[12], where

$$l(y, y') = \sum_i (y_i - y'_i)^2. \tag{3.3.2}$$

As we will see later, the L_2 loss corresponds to the assumption that our data was corrupted by Gaussian noise, whereas the L_1 loss corresponds to an assumption of noise from a Laplace distribution.

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 their mobile app. This would involve the customer snapping a photo of a check with their smart phone's camera and the machine learning model would need to be able to automatically understand text seen in the image. It would also need to understand hand-written text to be even more robust. This kind of system is referred to as optical character recognition (OCR), and the kind of problem it addresses 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 a feature vector, e.g. the pixel values in an image, and then predict which category (formally called *classes*), among some (discrete) set of options, an example belongs. For hand-written digits, we might have 10 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 X could consist of images of animals and our *labels* Y might be the classes $\{\text{cat}, \text{dog}\}$. While in regression, we sought a *regressor* to output a real value $\hat{y}$, in classification, we seek a *classifier*, whose output $\hat{y}$ is the predicted class assignment.

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 an example x, our model assigns a probability $\hat{y}_k$ to each label k. Because these are probabilities, they need to be positive numbers and add up to 1 and thus we only need $K - 1$ numbers to assign probabilities of K categories. This is easy to see for binary classification. If there is a 0.6 (60%) probability that an unfair coin comes up heads, then there is a 0.4 (40%) probability that it comes up tails. Returning to our animal classification example, a classifier might see an image and output the probability that the image is a cat $P(y = \text{cat}|x) = 0.9$. 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.

[11] http://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.L1Loss
[12] http://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.L2Loss

When we have more than two possible classes, we call the problem *multiclass classification*. Common examples include hand-written character recognition `[0, 1, 2, 3 ... 9, a, b, c, ...]`. While we attacked regression problems by trying to minimize the L1 or L2 loss functions, the common loss function for classification problems is called cross-entropy. In MXNet Gluon, the corresponding loss function can be found here[13].

Note that the most likely class is not necessarily the one that you are going to use for your decision. Assume that you find this beautiful mushroom in your backyard:

Fig. 3.3.2: 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 $P(y = \text{deathcap}|\text{image}) = 0.2$. In other words, the classifier is 80% sure that our mushroom *is not* a death cap. Still, you'd 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. We can look at this more formally. Basically, we need to compute the expected risk that we incur, i.e., we need to multiply the probability of the outcome with the benefit (or harm) associated with it:

$$L(\text{action}|x) = \mathbf{E}_{y \sim p(y|x)}[\text{loss}(\text{action}, y)] \tag{3.3.3}$$

Hence, the loss L incurred by eating the mushroom is $L(a = \text{eat}|x) = 0.2 * \infty + 0.8 * 0 = \infty$, whereas the cost of discarding it is $L(a = \text{discard}|x) = 0.2 * 0 + 0.8 * 1 = 0.8$.

Our caution was justified: as any mycologist would tell us, the above mushroom 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

[13] https://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.SoftmaxCrossEntropyLoss

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[14], who organized the animals in a hierarchy.

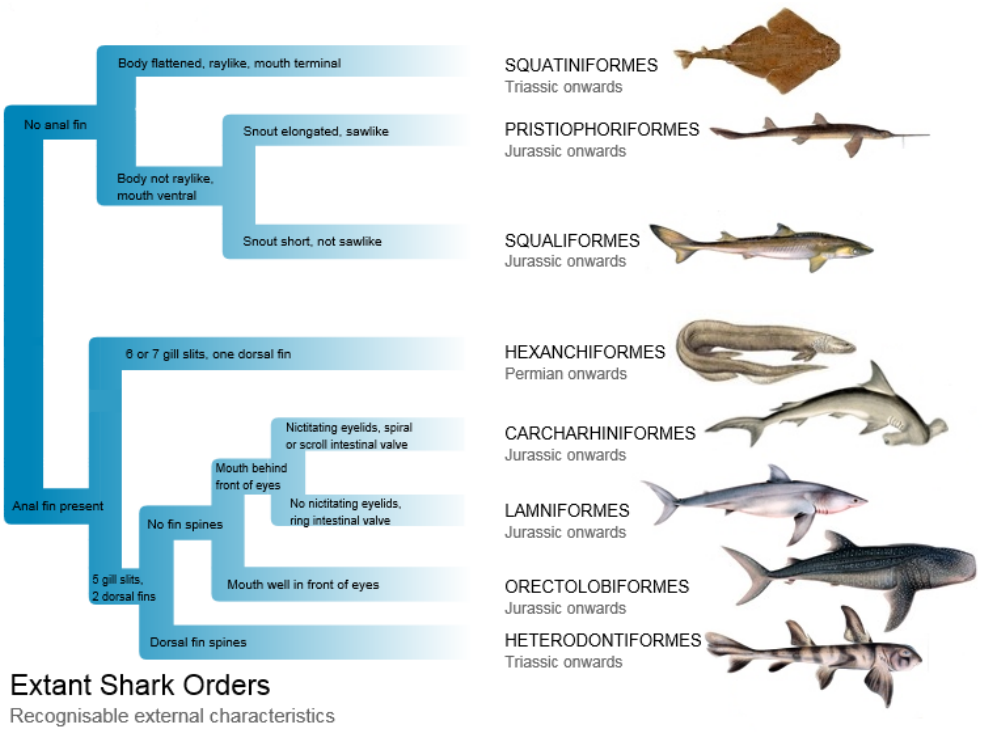


Fig. 3.3.3: Classify sharks

In the case of animal classification, it might not be so bad to mistake a poodle for a schnauzer, but our model would pay a huge penalty if it confused a poodle 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 do not 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.

As you can see, there is a cat in the picture, and a rooster, a dog, a donkey and a bird, 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 *and* a dog *and* a donkey *and* a rooster *and* a bird.

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 tech 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'.

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

Fig. 3.3.4: A cat, a roster, a dog and a donkey

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 28k 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 a competition[15] to do precisely this.

Search and ranking

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 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[16], 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 SciFi 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, or 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, GoodReads, etc.). 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 y_{ij}, such as an estimated rating or the probability of purchase, given a user u_i and product p_j.

Given such a model, then for any given user, we could retrieve the set of objects with the largest scores y_{ij}, which are could then be recommended to the customer. Production systems are considerably more advanced and take detailed user activity and item characteristics into account when computing such scores. The following image is an example of deep learning books recommended by Amazon based on personalization algorithms tuned to capture the author's preferences.

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: you might notice that items receive many 5 and 1 star ratings but that there are conspicuously few 3-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.

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

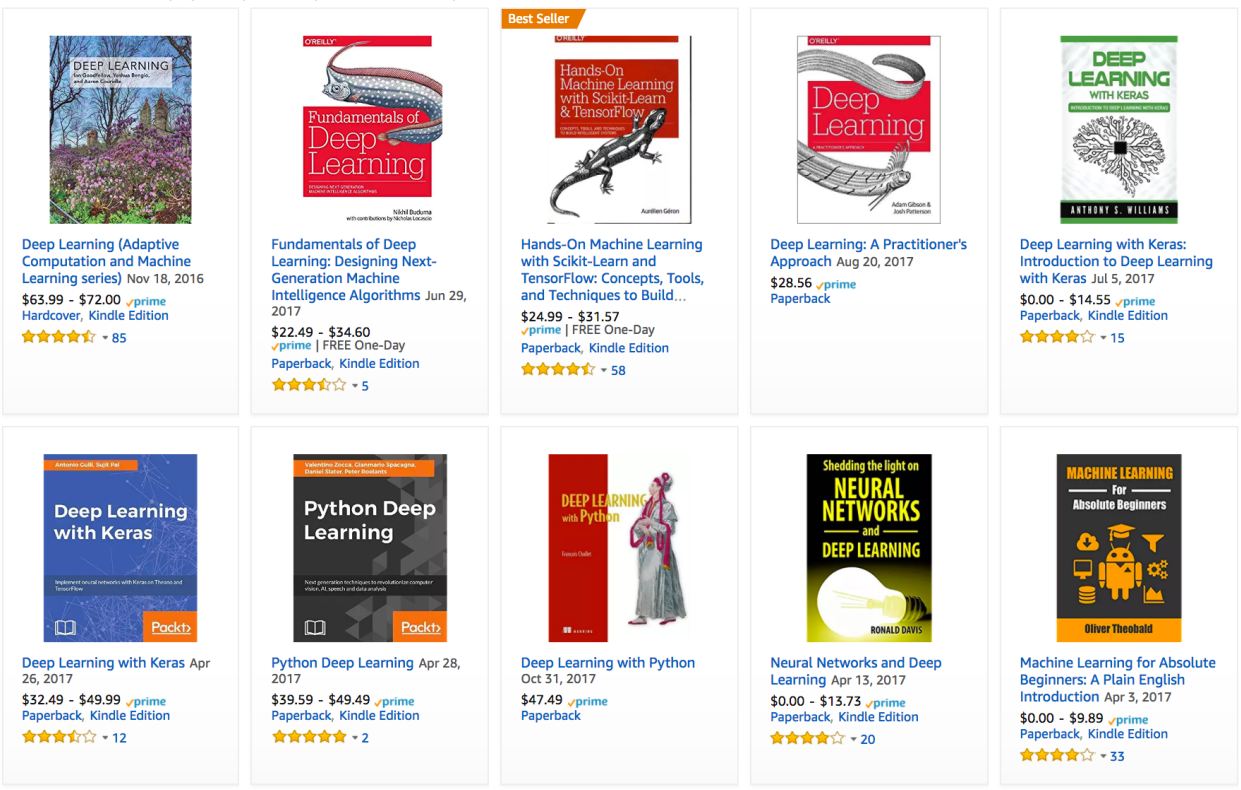


Fig. 3.3.5: Deep learning books recommended by Amazon.

Sequence Learning

So far, we have looked at problems where we have some fixed number of inputs and produce a fixed number of outputs. Before we considered predicting home 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!). These latter problems are sometimes referred to as `seq2seq` problems. Language translation is a `seq2seq` problem. Transcribing text from spoken speech is also a `seq2seq` problem. While it is impossible to consider all types of sequence transformations, a number of 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.

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

Automatic Speech Recognition

With speech recognition, the input sequence x is an audio recording of a speaker, and the output y 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 correspond to a single spoken word. These are `seq2seq` problems where the output is much shorter than the input.

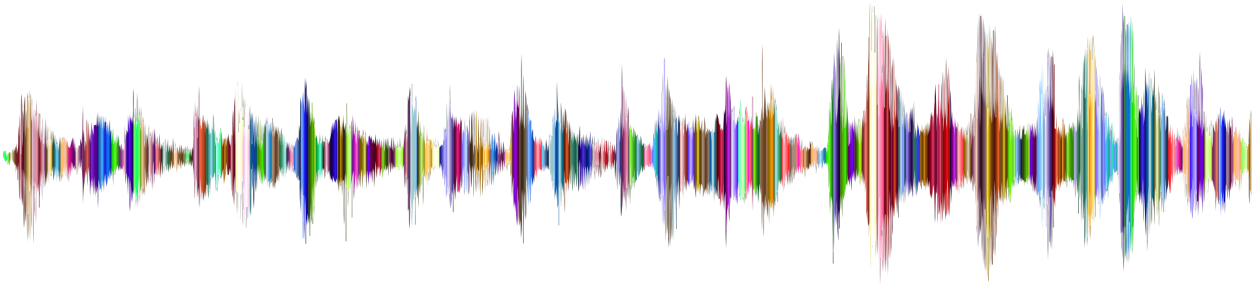


Fig. 3.3.6: `-D-e-e-p- L-ea-r-ni-ng-`

Text to Speech

Text-to-Speech (TTS) is the inverse of speech recognition. In other words, the input x is text and the output y 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 points 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. This is an active area of research.

3.3.2 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 target values. You could think of the supervised learner as having an extremely specialized job and an extremely anal 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'd 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 a number of unsupervised learning techniques in later chapters. To whet your appetite for now, we describe a few of the 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, mountain peaks, etc.? Likewise, given a collection of users' browsing activity, 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* problems. If the dependence is linear, it is called *principal component analysis*.
- Is there a representation of (arbitrarily structured) objects in Euclidean space (i.e., the space of vectors in $\mathbb{R}^n$) such that symbolic properties can be well matched? This is called *representation learning* and it is 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, salaries, etc., 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. We will devote a few notebooks to them.

3.3.3 Interacting with an Environment

So far, we have not discussed where data actually comes from, or what actually *happens* when a machine learning model generates an output. That is because supervised learning and unsupervised learning do not address these issues in a very sophisticated way. In either case, we grab a big pile of data up front, 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 looks like this:

This simplicity of offline learning has its charms. The upside is 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 of taking actions in the world. We want to think about intelligent *agents*,

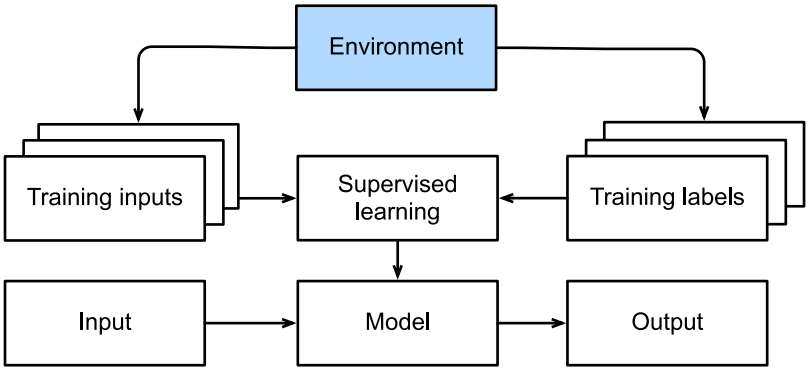


Fig. 3.3.7: Collect data for supervised learning from an environment.

not just predictive *models*. That means 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. Does the environment:

- Remember what we did previously?
- Want to help us, e.g., a user reading text into a speech recognizer?
- Want to beat us, i.e., an adversarial setting like spam filtering (against spammers) or playing a game (vs an opponent)?
- Not care (as in many cases)?
- Have shifting dynamics (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 homeworks were composed by her TAs. We will briefly describe reinforcement learning and adversarial learning, two settings that explicitly consider interaction with an environment.

3.3.4 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* (RL). This might include applications to robotics, to dialogue systems, and even to developing AI for video games. *Deep reinforcement learning* (DRL), which applies deep neural networks to RL problems, has surged in popularity. The breakthrough deep Q-network that beat humans at Atari games using only the visual input[17], and the AlphaGo program that dethroned the world champion at the board game Go[18] 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 t, the agent receives some observation o_t from the environment and must choose an action a_t that is subsequently transmitted back to the environment via some mechanism (sometimes called an actuator). Finally, the agent receives a reward r_t from the environment. The agent then receives a subsequent observation, and chooses a subsequent action, and so on. The behavior of an RL 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.

[17] https://www.wired.com/2015/02/google-ai-plays-atari-like-pros/
[18] https://www.wired.com/2017/05/googles-alphago-trounces-humans-also-gives-boost/

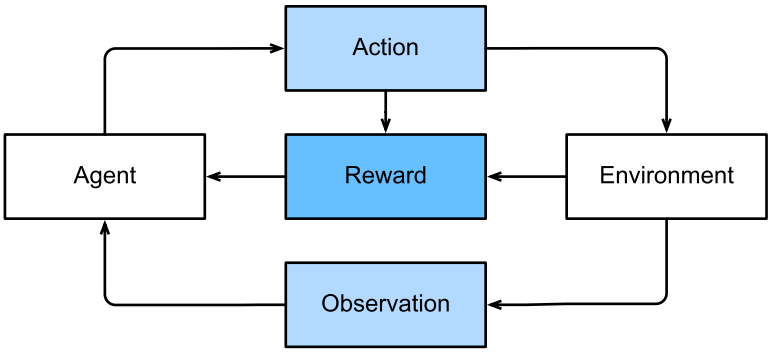


Fig. 3.3.8: The interaction between reinforcement learning and an environment.

It is hard to overstate the generality of the RL framework. For example, we can cast any supervised learning problem as an RL problem. Say we had a classification problem. We could create an RL 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 problem.

That being said, RL 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 RL, 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.

MDPs, bandits, and friends

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 RL problem a *Markov Decision Process* (MDP). 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*.

3.4 Roots

Although many deep learning methods are recent inventions, humans have held the desire to analyze data and to predict future outcomes for centuries. In fact, much of natural science has its roots in this. For instance, the Bernoulli distribution is named after Jacob Bernoulli (1655-1705)[19], and the Gaussian distribution was discovered by Carl Friedrich Gauss (1777-1855)[20]. 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)[21] illustrates averaging the length of 16 adult men's feet to obtain the average foot length.

Fig. 3.4.1: Estimating the length of a foot

Figure 1.1 illustrates how this estimator works. The 16 adult men were asked to line up in a row, when leaving 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.

Statistics really took off with the collection and availability of data. One of its titans, Ronald Fisher (1890-1962)[22], 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 (even the Iris dataset that he released in 1936 is still used sometimes to illustrate machine learning algorithms). Fisher was also a proponent of eugenics, which should remind us that the morally dubious use data science has as long and enduring a history as its productive use in industry and the natural sciences.

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

A second influence for machine learning came from Information Theory (Claude Shannon, 1916-2001)[23] and the Theory of computation via Alan Turing (1912-1954)[24]. Turing posed the question "can machines think?" in his famous paper Computing machinery and intelligence[25] (Mind, October 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)[26]. In his groundbreaking book The Organization of Behavior[27] (John Wiley & Sons, 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 (aka *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 due to a number of reasons. Training a network is computationally very expensive. While RAM was plentiful at the end of the past century, computational power was scarce. Secondly, datasets were relatively small. In fact, Fisher's 'Iris dataset' from 1932 was a popular tool for testing the efficacy of algorithms. MNIST with its 60,000 handwritten digits was considered huge.

Given the scarcity of data and computation, strong statistical tools such as Kernel Methods, Decision Trees and Graphical Models proved empirically superior. Unlike neural networks, they did not require weeks to train and provided predictable results with strong theoretical guarantees.

3.5 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 3.5.1.

Table 3.5.1: Dataset versus 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)

[23] https://en.wikipedia.org/wiki/Claude_Shannon
[24] https://en.wikipedia.org/wiki/Alan_Turing
[25] https://en.wikipedia.org/wiki/Computing_Machinery_and_Intelligence
[26] https://en.wikipedia.org/wiki/Donald_O._Hebb
[27] http://s-f-walker.org.uk/pubsebooks/pdfs/The_Organization_of_Behavior-Donald_O._Hebb.pdf

It is evident that RAM 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 needed 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 compute budget. Consequently the sweet spot in machine learning and statistics moved from (generalized) linear models and kernel methods to deep 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 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. (Bahdanau et al., 2014) found an elegant solution by using what can only be viewed as a learnable pointer structure. Rather than having to remember an entire sentence, 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 sentences, since the model no longer needed to remember the entire sentence before commencing the generation of a new sentence.
- Multi-stage designs, e.g., via the Memory Networks (MemNets) (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 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 GANS (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 GANs 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.
- 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 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 32k 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 64k observations, reducing training time for ResNet50 on ImageNet 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[28], Torch[29], and Theano[30]. Many seminal papers were written using these tools. By now, they have been superseded by TensorFlow[31], often used via its high level API Keras[32], CNTK[33], Caffe 2[34], and Apache MxNet[35]. The third generation of tools, namely imperative tools for deep learning, was arguably spearheaded by Chainer[36], which used a syntax similar to Python NumPy to describe models. This idea was adopted by PyTorch[37] and the Gluon API[38] of MXNet. It is the latter group that this course uses to teach deep learning.

The division of labor between systems researchers building better tools and statistical modelers building better 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 PhD 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.

3.6 Success Stories

Artificial Intelligence has a long history of delivering results that would be difficult to accomplish otherwise. For instance, mail is sorted using optical character recognition. These systems have been deployed since the 90s (this is, after all, the source of the famous MNIST and USPS sets 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, MasterCard. Computer programs for chess have been competitive for decades. Machine learning feeds search, recommendation, personalization and ranking on the Internet. In other words, artificial intelligence and machine learning are 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.

- Intelligent assistants, such as Apple's Siri, Amazon's Alexa, or 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 (Xiong et al., 2018) for certain applications.
- 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 (Lin et al., 2010) achieved a top-5 error rate of 28%. By 2017, (Hu et al., 2018) reduced this error rate to 2.25%. Similarly stunning results have been achieved for identifying birds, or diagnosing skin cancer.
- Games used to be a bastion of human intelligence. Starting from TDGammon [23], a program for playing Backgammon using temporal difference (TD) 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 Gary Kasparov, Campbell et al. (Campbell et al., 2002), using massive parallelism, special purpose hardware and efficient search through the game tree. Go is more difficult still, due to its huge state space. AlphaGo reached human parity in 2015, (Silver et al., 2016) using Deep Learning combined with Monte Carlo tree sampling. The challenge in Poker was that the state space is large and it is not fully observed (we do not

[28] https://github.com/BVLC/caffe
[29] https://github.com/torch
[30] https://github.com/Theano/Theano
[31] https://github.com/tensorflow/tensorflow
[32] https://github.com/keras-team/keras
[33] https://github.com/Microsoft/CNTK
[34] https://github.com/caffe2/caffe2
[35] https://github.com/apache/incubator-mxnet
[36] https://github.com/chainer/chainer
[37] https://github.com/pytorch/pytorch
[38] https://github.com/apache/incubator-mxnet

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 Momenta, Tesla, NVIDIA, MobilEye, 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. ML 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 cars, 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 syste that is ready to manipulate its human creators (or burn their coffee). Firstly, 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 (truck drivers and shop assistants are some of the most common jobs in many states). 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.

3.7 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, artificial intelligence, and optimization. Often, it is used as a means of implementing artificially-intelligent solutions.
- As a class of machine learning, representational learning focuses on how to automatically find the appropriate way to represent data. This is often accomplished by a progression of learned transformations.
- 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 good performance. The availability of efficient deep learning frameworks has made design and implementation of this significantly easier.

3.8 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 artificial intelligence 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? Physics? Engineering? Econometrics?

3.9 Scan the QR Code to Discuss[39]

[39] https://discuss.mxnet.io/t/2310

CHAPTER

FOUR

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 Section 18 should be sufficient.

4.1 Data Manipulation

In order to get anything done, we must have some way to 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 if we do not even know how to store it, so let us get our hands dirty first by playing with synthetic data. We will start by introducing the n-dimensional array (`ndarray`), MXNet's primary tool for storing and transforming data. In MXNet, `ndarray` is a class and we also call its instance an `ndarray` for brevity.

If you have worked with NumPy, perhaps the most widely-used scientific computing package in Python, then you are ready to fly. In short, we designed MXNet's `ndarray` to be an extension to NumPy's `ndarray` with a few key advantages. First, MXNet's `ndarray` supports asynchronous computation on CPU, GPU, and distributed cloud architectures, whereas the latter only supports CPU computation. Second, MXNet's `ndarray` supports automatic differentiation. These properties make MXNet's `ndarray` indispensable for deep learning. Throughout the book, the term `ndarray` refers to MXNet's `ndarray` unless otherwise stated.

4.1.1 Getting Started

Throughout this chapter, our aim is to get you up and running, equipping you with the the basic math and numerical computing tools that you will be mastering throughout the course of the book. Do not worry if you are not completely comfortable with all of the mathematical concepts or library functions. In the following sections we will revisit the same material in the context practical examples. 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 the same 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 `ndarray`, we almost always invoke the `set_np` function: this is for compatibility of `ndarray` processing by other components of MXNet.

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

An `ndarray` represents an array of numerical values, which are possibly multi-dimensional. With one axis, an `ndarray` corresponds (in math) to a *vector*. With two axes, an `ndarray` corresponds to a *matrix*. Arrays with more than two axes do not have special mathematical names—we simply call them *tensors*.

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 an `ndarray` is called an *element* of the `ndarray`. For instance, there are 12 elements in the `ndarray` `x`. Unless otherwise specified, a new `ndarray` 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 an `ndarray`'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 an `ndarray`, i.e., the product of all of the shape elements, we can inspect its `size` property. 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 an `ndarray` without altering either the number of elements or their values, we can invoke the `reshape` function. For example, we can transform our `ndarray`, `x`, from a row vector with shape (12,) to a matrix of shape (3, 4). This new `ndarray` contains the exact same values, and treats such values as a matrix organized as 3 rows and 4 columns. To reiterate, although the shape has changed, the elements in `x` have not. Consequently, the `size` remains the same.

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

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

Reshaping by manually specifying each of the dimensions can sometimes get annoying. For instance, if our target shape is a matrix with shape (height, width), 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, `ndarray` can automatically work out one dimension given the rest. We invoke this capability by placing `-1` for the dimension that we would like `ndarray` 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)`.

The `empty` method grabs a chunk of memory and hands us back a matrix without bothering to change the value of any of its entries. This is remarkably efficient but we must be careful because the entries might take arbitrary values, including very big ones!

```
np.empty((3, 4))
```

```
array([[1.3700303e-09, 4.5861696e-41, 7.0707049e+26, 3.0911243e-41],
       [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00],
       [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00]])
```

Typically, we will want our matrices initialized either with ones, zeros, some known constants, or numbers randomly sampled from a known distribution. Perhaps most often, we want an array of all zeros. To create an `ndarray` representing a tensor with all elements set to 0 and a shape of (2, 3, 4) we can invoke

```
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.]]])
```

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.]]])
```

In some cases, we will want to randomly sample the values of all the elements in an `ndarray` according to some known probability distribution. One common case is when we construct an array to serve as a parameter in a neural network. The following snippet creates an `ndarray` 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 ,  0.7740038 ,  1.0434405 ,  1.1839255 ],
       [ 1.8917114 , -1.2347414 , -1.771029  , -0.45138445],
       [ 0.57938355, -1.856082  , -1.9768796 , -0.20801921]])
```

We can also specify the value of each element in the desired `ndarray` by supplying a Python list containing the numerical values.

```
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.]])
```

4.1.2 Operations

This book is not about Web development—it is not enough to just read and write values. 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 math notation, we would denote such a *unary* scalar operator (taking one input) by the signature $f : \mathbb{R} \rightarrow \mathbb{R}$ and a *binary* scalar operator (taking two inputs) 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.

In MXNet, 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.

```
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 4.4.

We can also *concatenate* multiple ndarrays together, stacking them end-to-end to form a larger ndarray. We just need to provide a list of ndarrays 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 ndarray's axis-0 length (6) is the sum of the two input ndarrays' axis-0 lengths ($3 + 3$); while the second output ndarray's axis-1 length (8) is the sum of the two input ndarrays' 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 ndarray 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 ndarray 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([[0., 1., 0., 1.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

Summing all the elements in the ndarray yields an ndarray with only one element.

```
x.sum()
```

```
array(66.)
```

For stylistic convenience, we can write x.sum() as np.sum(x).

4.1.3 Broadcasting Mechanism

In the above section, we saw how to perform elementwise operations on two ndarrays of the same shape. Under certain conditions, even when shapes differ, we can still perform elementwise operations by invoking the *broadcasting mechanism*. These mechanisms work in the following way: First, expand one or both arrays by copying elements appropriately so that after this transformation, the two ndarrays 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.]])
```

4.1.4 Indexing and Slicing

Just as in any other Python array, elements in an `ndarray` 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.

By this logic, `[-1]` selects the last element and `[1:3]` selects the second and the third elements. Let us try this out and compare the outputs.

```
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.]])
```

4.1.5 Saving Memory

In the previous example, every time we ran an operation, we allocated new memory to host its results. For example, if we write `y = x + y`, we will dereference the `ndarray` 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 redirects `y` to point at 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, this could cause that discarded memory is not released, and make it possible for parts of our code to inadvertently reference stale parameters.

Fortunately, performing in-place operations in MXNet 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): 140562856940880
id(z): 140562856940880
```

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
```

4.1.6 Conversion to Other Python Objects

Converting an MXNet's `ndarray` to an object in the NumPy package of Python, 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 MXNet 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. The `array` and `asnumpy` functions do the trick.

```
a = x.asnumpy()
b = np.array(a)
type(a), type(b)
```

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

To convert a size-1 `ndarray` 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)
```

4.1.7 Summary

- MXNet's `ndarray` is an extension to NumPy's `ndarray` with a few key advantages that make the former indispensable for deep learning.
- MXNet's `ndarray` provides a variety of functionalities such as basic mathematics operations, broadcasting, indexing, slicing, memory saving, and conversion to other Python objects.

4.1.8 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 `ndarray` you can get.
2. Replace the two `ndarrays` that operate by element in the broadcasting mechanism with other shapes, e.g., three dimensional tensors. Is the result the same as expected?

4.1.9 Scan the QR Code to Discuss[40]

4.2 Data Preprocessing

So far we have introduced a variety of techniques for manipulating data that are already stored in `ndarrays`. To apply deep learning to solving real-world problems, we often begin with preprocessing raw data, rather than those nicely prepared data in the `ndarray` 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 `ndarray`. So, we will briefly walk through steps for preprocessing raw data with `pandas` and converting them into the `ndarray` format. We will cover more data preprocessing techniques in later chapters.

[40] https://discuss.mxnet.io/t/2316

4.2.1 Loading Data

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

```
# Write the dataset row by row into a csv file
data_file = '../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 is a data point
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

To load the raw dataset from the created csv file, we import the `pandas` package and invoke the `read_csv` function. This dataset has 4 rows and 3 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
```

4.2.2 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 2 columns while the later 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 2 types of categorical values "Pave" and "NaN", `pandas` can automatically convert this column to 2 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
```

4.2.3 Conversion to the `ndarray` Format

Now that all the entries in `inputs` and `outputs` are numerical, they can be converted to the `ndarray` format. Once data are in this format, they can be further manipulated with those `ndarray` functionalities that we have introduced in Section 4.1.

```
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.]]), array([127500., 106000., 178100., 140000.]))
```

4.2.4 Summary

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

4.2.5 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 `ndarray` format.

4.2.6 Scan the QR Code to Discuss[41]

[41] https://discuss.mxnet.io/t/4973

4.3 Scalars, Vectors, Matrices, and Tensors

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 in linear algebra, expressing each both through mathematical notation and the corresponding implementation in code.

4.3.1 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 represented 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.

In MXNet code, a scalar is represented by an `ndarray` 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.))
```

4.3.2 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 $\mathbf{z}$).

In MXNet, we work with vectors via 1-dimensional `ndarrays`. In general `ndarrays` 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

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{4.3.1}$$

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

```
x[3]
```

```
array(3.)
```

Length, Dimensionality, and Shape

Let us revisit some concepts from Section 4.1. 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 an `ndarray` by calling Python's built-in `len()` function.

```
len(x)
```

```
4
```

When an `ndarray` 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 `ndarray`. For `ndarrays` 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 an `ndarray` to refer to the number of axes that an `ndarray` has. In this sense, the dimensionality of an `ndarray`'s some axis will be the length of that axis.

4.3.3 Matrices

Just as vectors generalize scalars from order 0 to order 1, matrices generalize vectors from order 1 to order 2. Matrices, which we will typically denote with bold-faced, capital letters (e.g., $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$), are represented in code as `ndarrays` with 2 axes.

In math notation, we use $\mathbf{A} \in \mathbb{R}^{m \times n}$ to express that the matrix $\mathbf{A}$ 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:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{4.3.2}$$

For any $\mathbf{A} \in \mathbb{R}^{m \times n}$, the shape of $\mathbf{A}$ 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 in MXNet by specifying a shape with two components m and n when calling any of our favorite functions for instantiating an `ndarray`.

```
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 $\mathbf{A}$ in (4.3.2) by specifying the indices for the row (i) and column (j), such as $[\mathbf{A}]_{ij}$. When the scalar elements of a matrix $\mathbf{A}$, such as in (4.3.2), are not given, we may simply use the lower-case letter of the matrix $\mathbf{A}$ 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 $\mathbf{A}$'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 $\mathbf{A}$ in (4.3.2) is a $n \times m$ matrix:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}. \tag{4.3.3}$$

In code, we access a matrix's transpose via the `T` attribute.

```
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: $\mathbf{A} = \mathbf{A}^\top$.

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

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

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

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

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 points), while columns might correspond to different attributes. This should sound familiar if you have ever used spreadsheet software or have read Section 4.2. 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 point 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 an `ndarray`, we can access or enumerate minibatches of data points, or just data points if no minibatch exists.

4.3.4 Tensors

Just as vectors generalize scalars, and matrices generalize vectors, we can build data structures with even more axes. Tensors give us a generic way of describing `ndarrays` 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 `ndarrays` 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.],
        [16., 17., 18., 19.],
        [20., 21., 22., 23.]]])
```

4.3.5 Summary

- Scalars, vectors, matrices, and tensors are basic mathematical objects in linear algebra.
- Vectors generalize scalars, and matrices generalize vectors.
- In the `ndarray` representation, scalars, vectors, matrices, and tensors have 0, 1, 2, and an arbitrary number of axes, respectively.

4.3.6 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?

4.3.7 Scan the QR Code to Discuss[42]

4.4 Reduction, Multiplication, and Norms

We have introduced scalars, vectors, matrices, and tensors in Section 4.3. Without operations on such basic mathematical objects in linear algebra, we cannot design any meaningful procedure for deep learning. Specifically, as we will see in subsequent chapters, reduction, multiplication, and norms are arguably among the most commonly used operations. In this section, we start off by reviewing and summarizing basic properties of tensor arithmetic, then illustrate those operations in both mathematics and code.

4.4.1 Basic Properties of Tensor Arithmetic

Scalars, vectors, matrices, and tensors 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.

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

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.]]))
```

[42] https://discuss.mxnet.io/t/2317

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 (4.3.2)) and $\mathbf{B}$

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \tag{4.4.1}$$

```
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))
```

4.4.2 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 `sum` function.

```
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 `sum` function *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 `sum`. 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 `mean` on tensors of arbitrary shape.

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

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

Like `sum`, `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 `sum` or `mean` by setting `keepdims=True`.

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

```
array([[ 6.],
       [22.],
       [38.],
```

(continues on next page)

(continued from previous page)

```
       [54.],
       [70.]])
```

For instance, since `sum_A` still keeps its 2 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.]])
```

4.4.3 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.

4.4.4 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 (4.3.2) and (4.3.1) respectively. Let us start off by visualizing the matrix $\mathbf{A}$ in terms of its row vectors

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \tag{4.4.2}$$

where each $\mathbf{a}_i^\top \in \mathbb{R}^n$ is a row vector representing the i^{th} row of the matrix $\mathbf{A}$. The matrix-vector product $\mathbf{Ax}$ is simply a column vector of length m, whose i^{th} element is the dot product $\mathbf{a}_i^\top \mathbf{x}$:

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}. \tag{4.4.3}$$

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 `ndarrays`, 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.]))
```

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

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}. \tag{4.4.4}$$

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:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix}. \tag{4.4.5}$$

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 $\mathbf{a}_i^\top \mathbf{b}_j$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}. \tag{4.4.6}$$

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. Just as with ordinary dot products and matrix-vector products, we can compute matrix-matrix multiplication by using the `dot` function. 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.

4.4.6 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 **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:

$$f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x}). \tag{4.4.7}$$

The second property is the familiar triangle inequality:

$$f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}). \tag{4.4.8}$$

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

$$f(\mathbf{x}) \geq 0. \tag{4.4.9}$$

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.

$$\forall i, [\mathbf{x}]_i = 0 \Leftrightarrow f(\mathbf{x}) = 0. \tag{4.4.10}$$

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 **x** are $x_1, \ldots, x_n$. The ℓ_2 *norm* of **x** is the square root of the sum of the squares of the vector elements:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}, \tag{4.4.11}$$

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 by calling `linalg.norm`.

```
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:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| \,. \tag{4.4.12}$$

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*:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}. \tag{4.4.13}$$

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:

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}^2}. \tag{4.4.14}$$

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 `linalg.norm` 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.

4.4.7 More on Linear Algebra

In just Section 4.3 and 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 Section 18.1 or other excellent resources (Strang, 1993; Kolter, 2008; Petersen et al., 2008).

4.4.8 Summary

- 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 with `ndarray` functions.

4.4.9 Exercises

1. Run `A / A.sum(axis=1)` and see what happens. Can you analyze the reason?
2. 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?
3. Consider a tensor with shape $(2, 3, 4)$. What are the shapes of the summation outputs along axis 0, 1, and 2?
4. Feed a tensor with 3 or more axes to the `linalg.norm` function and observe its output. What does this function compute for `ndarrays` of arbitrary shape?

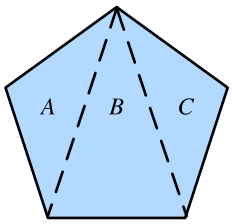
4.4.10 Scan the QR Code to Discuss[43]

4.5 Calculus

At least 2,500 years ago, ancient Greeks were able find the area of a polygon: divide the polygon into triangles, then sum the areas of these triangles. Fig. 4.5.1 illustrates this method.

To find the area of curved shapes, such as a circle, ancient Greeks inscribed polygons in such shapes. As shown in Fig. 4.5.1, an inscribed polygon with more sides of equal length better approximates the circle.

[43] https://discuss.mxnet.io/t/4974



Area = $A + B + C$

Fig. 4.5.1: Find the area of a polygon.

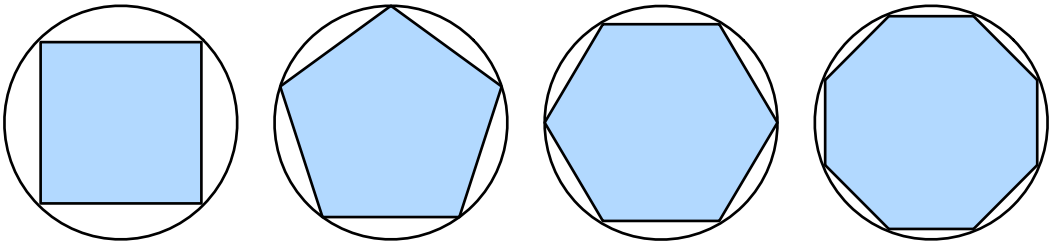


Fig. 4.5.2: Find the area of the circle.

The process in Fig. 4.5.2 is called *method of exhaustion*. This 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 4.4.6, 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 points 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.

4.5.1 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

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}, \tag{4.5.1}$$

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 (4.5.1) 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
import d2l
from IPython import display
from mxnet import np, npx
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 (4.5.1) 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('h=%.5f, numerical limit=%.5f' % (h, numerical_lim(f, 1, h)))
    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:

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx}f(x) = Df(x) = D_x f(x), \tag{4.5.2}$$

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*

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x), \tag{4.5.3}$$

the *sum rule*

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \tag{4.5.4}$$

the *product rule*

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \tag{4.5.5}$$

and the *quotient rule*

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \tag{4.5.6}$$

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. The comment `# Saved in the d2l package for later use` is a special mark where the following function, class, or import statements are also saved in the `d2l` package so that we can directly invoke `d2l.use_svg_display()` later.

```
# Saved in the d2l package for later use
def use_svg_display():
    """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.

```
# Saved in the d2l package for later use
def set_figsize(figsize=(3.5, 2.5)):
    """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`.

```
# Saved in the d2l package for later use
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 3 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.

```
# Saved in the d2l package for later use
def plot(X, Y=None, xlabel=None, ylabel=None, legend=[], xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=['-', 'm--', 'g-.', 'r:'], figsize=(3.5, 2.5), axes=None):
    """Plot data points."""
    d2l.set_figsize(figsize)
    axes = axes if axes else d2l.plt.gca()
```

(continues on next page)

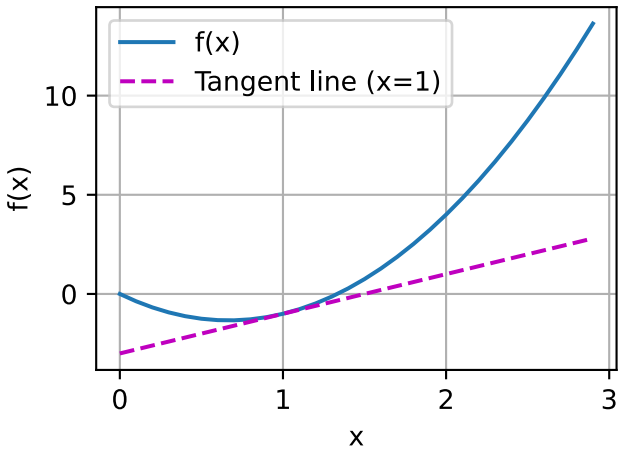
(continued from previous page)

```
    # Return True if X (ndarray 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)'])
```



4.5.2 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

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{4.5.7}$$

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:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f. \tag{4.5.8}$$

4.5.3 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:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right]^\top, \tag{4.5.9}$$

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

Let $\mathbf{A}$ be a matrix with m rows and n columns, and $\mathbf{x}$ be an n-dimensional vector, the following rules are often used when differentiating multivariate functions:

- $\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$,
- $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$,
- $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$,
- $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$.

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.

4.5.4 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

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}. \tag{4.5.10}$$

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

$$\frac{dy}{dx_i} = \frac{dy}{du_1}\frac{du_1}{dx_i} + \frac{dy}{du_2}\frac{du_2}{dx_i} + \cdots + \frac{dy}{du_m}\frac{du_m}{dx_i} \tag{4.5.11}$$

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

4.5.5 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.

4.5.6 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 $z = z(a, b)$?

4.5.7 Scan the QR Code to Discuss[44]

4.6 Automatic Differentiation

As we have explained in Section 4.5, 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).

The `autograd` package expedites this work by automatically calculating derivatives, i.e., *automatic differentiation*. And while many other libraries require that we compile a symbolic graph to take automatic derivatives, `autograd` allows us to take derivatives while writing ordinary imperative code. Every time we pass data through our model, `autograd` builds a graph on the fly, tracking which data combined through which operations to produce the output. This graph enables `autograd` to subsequently backpropagate gradients on command. Here, *backpropagate* simply means to trace through the *computational graph*, filling in the partial derivatives with respect to each parameter.

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

4.6.1 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.

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

[44] https://discuss.mxnet.io/t/5008

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

Note that before we even calculate the gradient of y with respect to **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 also that a gradient of a scalar-valued function with respect to a vector **x** is itself vector-valued and has the same shape as **x**. Thus it is intuitive that in code, we will access a gradient taken with respect to `x` as an attribute of the `ndarray` `x` itself. We allocate memory for an `ndarray`'s gradient by invoking its `attach_grad` method.

```
x.attach_grad()
```

After we calculate a gradient taken with respect to `x`, we will be able to access it via the `grad` attribute. As a safe default, `x.grad` is initialized as an array containing all zeros. That is sensible because our most common use case for taking gradient in deep learning is to subsequently update parameters by adding (or subtracting) the gradient to maximize (or minimize) the differentiated function. By initializing the gradient to an array of zeros, we ensure that any update accidentally executed before a gradient has actually been calculated will not alter the parameters' value.

```
x.grad
```

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

Now let us calculate y. Because we wish to subsequently calculate gradients, we want MXNet to generate a computational graph on the fly. We could imagine that MXNet would be turning on a recording device to capture the exact path by which each variable is generated.

Note that building the computational graph requires a nontrivial amount of computation. So MXNet will only build the graph when explicitly told to do so. We can invoke this behavior by placing our code inside an `autograd.record` scope.

```
with autograd.record():
    y = 2 * np.dot(x, x)
y
```

```
array(28.)
```

Since `x` is an `ndarray` of length 4, `np.dot` will perform an inner product of `x` and `x`, 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 `y`'s `backward` function.

```
y.backward()
```

If we recheck the value of `x.grad`, we will find its contents overwritten by the newly calculated gradient.

```
x.grad
```

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

The gradient of the function $y = 2\mathbf{x}^\top\mathbf{x}$ with respect to **x** should be $4\mathbf{x}$. Let us quickly verify that our desired gradient was calculated correctly. If the two `ndarray`s are indeed the same, then the equality between them holds at every position.

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

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

If we subsequently compute the gradient of another variable whose value was calculated as a function of `x`, the contents of `x.grad` will be overwritten.

```
with autograd.record():
    y = x.sum()
y.backward()
x.grad
```

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

4.6.2 Backward for Non-Scalar Variables

Technically, when `y` is not a scalar, the most natural interpretation of the gradient of `y` (a vector of length m) with respect to `x` (a vector of length n) is the *Jacobian* (an $m \times n$ matrix). For higher-order and higher-dimensional `y` and `x`, the Jacobian could be a gnarly 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 Jacobian but rather the sum of the partial derivatives computed individually for each example in the batch.

Thus when we invoke `backward` on a vector-valued variable `y`, which is a function of `x`, MXNet assumes that we want the sum of the gradients. In short, MXNet will create a new scalar variable by summing the elements in `y`, and compute the gradient of that scalar variable with respect to `x`.

```
with autograd.record():
    y = x * x  # y is a vector
y.backward()

u = x.copy()
u.attach_grad()
with autograd.record():
    v = (u * u).sum()  # v is a scalar
v.backward()

x.grad == u.grad
```

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

4.6.3 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 call `u = y.detach()` 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`. This will provide the same functionality as if we had calculated `u` as a function of `x` outside of the `autograd.record` scope, yielding a `u` that will be treated as a constant in any `backward` call. Thus, the following `backward` 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([1., 1., 1., 1.])
```

Since the computation of `y` was recorded, we can subsequently call `y.backward()` to get the derivative of `y = x * x` with respect to `x`, which is `2 * x`.

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

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

Note that attaching gradients to a variable `x` implicitly calls `x = x.detach()`. If `x` is computed based on other variables, this part of computation will not be used in the `backward` function.

```
y = np.ones(4) * 2
y.attach_grad()
with autograd.record():
    u = x * y
    u.attach_grad()  # Implicitly run u = u.detach()
    z = 5 * u - x
z.backward()
x.grad, u.grad, y.grad
```

```
(array([-1., -1., -1., -1.]), array([5., 5., 5., 5.]), array([0., 0., 0., 0.]))
```

4.6.4 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
```

(continues on next page)

(continued from previous page)

```
    else:
        c = 100 * b
    return c
```

Again to compute gradients, we just need to `record` the calculation and then call the `backward` function.

```
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(1.)
```

4.6.5 Training Mode and Prediction Mode

As we have seen, after we call `autograd.record`, MXNet logs the operations in the following block. There is one more subtle detail to be aware of. Additionally, `autograd.record` will change the running mode from *prediction mode* to *training mode*. We can verify this behavior by calling the `is_training` function.

```
print(autograd.is_training())
with autograd.record():
    print(autograd.is_training())
```

```
False
True
```

When we get to complicated deep learning models, we will encounter some algorithms where the model behaves differently during training and when we subsequently use it to make predictions. We will cover these differences in detail in later chapters.

4.6.6 Summary

- MXNet provides the `autograd` package to 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 `backward` function, and access the resulting gradient via our variable's `grad` attribute.
- We can detach gradients to control the part of the computation that will be used in the `backward` function.
- The running modes of MXNet include training mode and prediction mode. We can determine the running mode by calling the `is_training` function.

4.6.7 Exercises

1. Why is the second derivative much more expensive to compute than the first derivative?
2. After running `y.backward()`, 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 $f'(x) = \cos(x)$.
6. In a second-price auction (such as in eBay or in computational advertising), the winning bidder pays the second-highest price. Compute the gradient of the final price with respect to the winning bidder's bid using `autograd`. What does the result tell you about the mechanism? If you are curious to learn more about second-price auctions, check out the paper by Edelman et al. :cite“Edelman.Ostrovsky.Schwarz.2007“.

4.6.8 Scan the QR Code to Discuss[45]

4.7 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 action. 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.

As shown in Fig. 4.7.1, 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 $P(y =$ “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

[45] https://discuss.mxnet.io/t/2318

Fig. 4.7.1: Images of varying resolutions (10×10, 20×20, 40×40, 80×80, and 160×160 pixels).

$P(y = \text{“cat”}) = P(y = \text{“dog”}) = 0.5$. 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.

4.7.1 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 6 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 d2l
from mxnet import np, npx
import random
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. In MXNet, we can sample from the multinomial distribution via the aptly named `np.random.multinomial` function. The function can be called in many ways, but we will focus on the simplest. To draw a single sample, we simply pass in a vector of probabilities. The output of the `np.random.multinomial` function 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 `random.multinomial` 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)
```

We can also conduct, say 3, groups of experiments, where each group draws 10 samples, all at once.

```
counts = np.random.multinomial(10, fair_probs, size=3)
counts
```

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

```
# Store the results as 32-bit floats for division
counts = np.random.multinomial(1000, fair_probs).astype(np.float32)
counts / 1000  # Reletive frequency as the estimate
```

```
array([0.164, 0.153, 0.181, 0.163, 0.163, 0.176])
```

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.

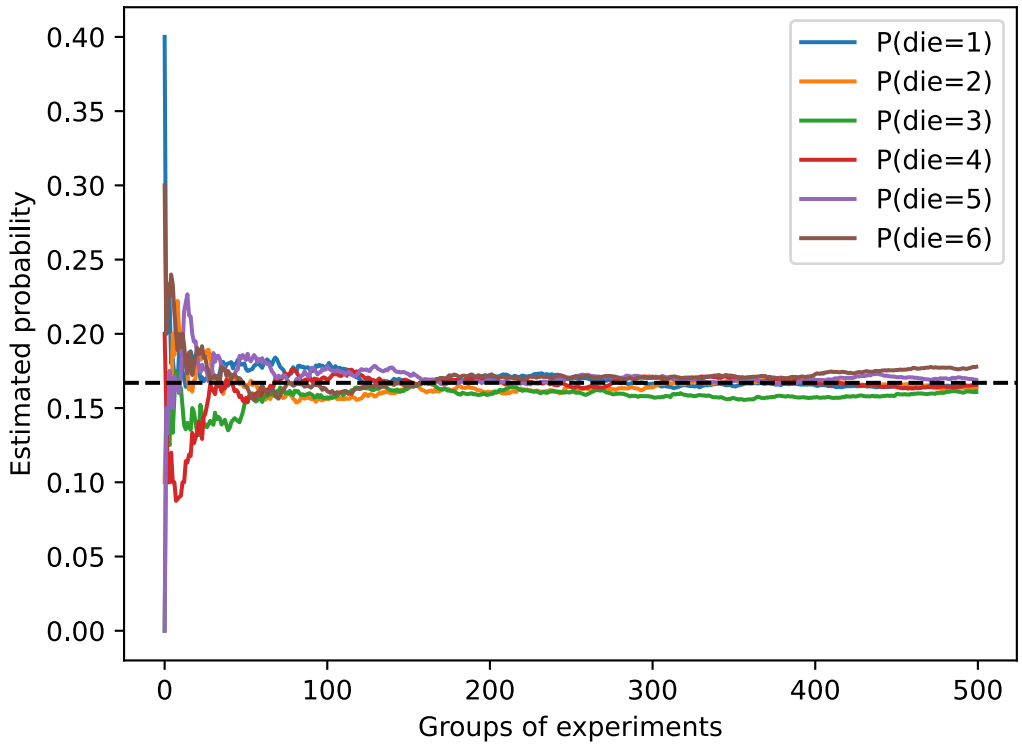
```
counts = np.random.multinomial(10, fair_probs, size=500)
cum_counts = counts.astype(np.float32).cumsum(axis=0)
estimates = cum_counts / cum_counts.sum(axis=1, keepdims=True)

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 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.

Random Variables

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.

4.7.2 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

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}. \tag{4.7.1}$$

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:

$$P(B) = \sum_A P(A, B), \tag{4.7.2}$$

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 are 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 AIDS 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 4.7.1 lists such conditional probability.

Table 4.7.1: 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 AIDS 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

$$\begin{aligned} & P(D_1 = 1) \\ =& P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1) \\ =& P(D_1 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1 \mid H = 1)P(H = 1) \\ =& 0.011485. \end{aligned} \tag{4.7.3}$$

Thus, we get

$$\begin{aligned}&P(H = 1 \mid D_1 = 1)\\ =&\frac{P(D_1 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1)}.\\ =&0.1306\end{aligned} \tag{4.7.4}$$

In other words, there is only a 13.06% chance that the patient actually has AIDS, despite using a very accurate test. As we can see, probability can be quite 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 4.7.2.

Table 4.7.2: 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:

$$\begin{aligned}&P(D_1 = 1, D_2 = 1 \mid H = 0)\\ =&P(D_1 = 1 \mid H = 0)P(D_2 = 1 \mid H = 0)\\ =&0.0003,\end{aligned} \tag{4.7.5}$$

$$\begin{aligned}&P(D_1 = 1, D_2 = 1 \mid H = 1)\\ =&P(D_1 = 1 \mid H = 1)P(D_2 = 1 \mid H = 1)\\ =&0.98.\end{aligned} \tag{4.7.6}$$

Now we can apply marginalization and the multiplication rule:

$$\begin{aligned}&P(D_1 = 1, D_2 = 1)\\ =&P(D_1 = 1, D_2 = 1, H = 0) + P(D_1 = 1, D_2 = 1, H = 1)\\ =&P(D_1 = 1, D_2 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1)\\ =&0.00176955.\end{aligned} \tag{4.7.7}$$

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

$$\begin{aligned}&P(H = 1 \mid D_1 = 1, D_2 = 1)\\ =&\frac{P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1, D_2 = 1)}\\ =&0.8307.\end{aligned} \tag{4.7.8}$$

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.

4.7.3 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

$$E[X] = \sum_x xP(X = x). \tag{4.7.9}$$

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

$$E_{x\sim P}[f(x)] = \sum_x f(x)P(x). \tag{4.7.10}$$

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

$$\text{Var}[X] = E\left[(X - E[X])^2\right] = E[X^2] - E[X]^2. \tag{4.7.11}$$

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:

$$\text{Var}[f(x)] = E\left[\left(f(x) - E[f(x)]\right)^2\right]. \tag{4.7.12}$$

4.7.4 Summary

- We can use MXNet to 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.

4.7.5 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[46].)
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[47].)
4. In Section 4.7.2, the first test is more accurate. Why not just run the first test a second time?

4.7.6 Scan the QR Code to Discuss[48]

[46] https://en.wikipedia.org/wiki/Venn_diagram
[47] https://en.wikipedia.org/wiki/Markov_chain
[48] https://discuss.mxnet.io/t/2319

4.8 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.

4.8.1 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 `np.random` module as follows:

```
from mxnet import np
print(dir(np.random))
```

```
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__
↪name__', '__package__', '__spec__', '_mx_nd_np', 'absolute_import', 'choice',
↪'multinomial', 'normal', 'randint', 'shuffle', 'uniform']
```

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`).

4.8.2 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 `ndarray`'s `ones_like` function.

```
help(np.ones_like)
```

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

ones_like(a=None, out=None, name=None, **kwargs)
    ones_like(a)

    Return an array of ones with the same shape and type as a given array.

    Parameters
    ----------
    a : ndarray
        The shape and data-type of a define these same attributes of
        the returned array.

    Returns
    -------
    out : ndarray
        Array of ones with the same shape and type as a.
```

From the documentation, we can see that the `ones_like` function creates a new array with the same shape as the supplied `ndarray` and sets all the elements to `1`. Whenever possible, you should run a quick test to confirm your interpretation:

```
x = np.array([[0, 0, 0], [2, 2, 2]])
np.ones_like(x)
```

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

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

4.8.3 API Documentation

For further details on the API details check the MXNet website at http://mxnet.apache.org/. You can find the details under the appropriate headings (also for programming languages other than Python).

4.8.4 Summary

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

4.8.5 Exercises

1. Look up `ones_like` and `autograd` on the MXNet website.
2. What are all the possible outputs after running `np.random.choice(4, 2)`?
3. Can you rewrite `np.random.choice(4, 2)` by using the `np.random.randint` function?

4.8.6 Scan the QR Code to Discuss[49]

[49] https://discuss.mxnet.io/t/2322

CHAPTER

FIVE

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 architecures, 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 logistic regression can be cast as *shallow* neural networks. Starting from these classic algorithms, we will introduce you to the basics, providing the basis for more complex techniques such as softmax regression (introduced at the end of this chapter) and multilayer perceptrons (introduced in the next chapter).

5.1 Linear Regression

Regression refers to a set of methods for modeling the relationship between data points **x** and corresponding real-valued targets y. 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.

5.1.1 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 *features* **x** and targets y is linear, i.e., that y can be expressed as a weighted sum of the inputs **x**, give or take 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 fit 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 data* or *training set*, and each row (here the data corresponding to one sale) is called an *instance* or *example*. The thing we are trying to predict (here, the price) is called a *target* or *label*.
The variables (here *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 samples by i, denoting each input data point as $x^{(i)} = [x_1^{(i)}, x_2^{(i)}]$ 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):

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b. \tag{5.1.1}$$

Here, 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 intercept or else we will limit the expressivity of our linear model.

Given a dataset, our goal is to choose the weights w and bias b such that on average, the predictions made according our model best fit the true prices observed in the data.

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 ML, 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 $\hat{y}$ as

$$\hat{y} = w_1 \cdot x_1 + ... + w_d \cdot x_d + b. \tag{5.1.2}$$

Collecting all features into a vector $\mathbf{x}$ and all weights into a vector $\mathbf{w}$, we can express our model compactly using a dot product:

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b. \tag{5.1.3}$$

Here, the vector $\mathbf{x}$ corresponds to a single data point.
We will often find it convenient to refer to our entire dataset via the *design matrix* X. Here, X contains one row for every example and one column for every feature.

For a collection of data points $\mathbf{X}$, the predictions $\hat{\mathbf{y}}$ can be expressed via the matrix-vector product:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b. \tag{5.1.4}$$

Given a training dataset X and corresponding (known) targets $\mathbf{y}$, the goal of linear regression is to find the *weight* vector w and bias term b that given some a new data point $\mathbf{x}_i$, sampled from the same distribution as the training data will (in expectation) predict the target y_i 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 real-world data where y_i exactly equals $\mathbf{w}^T \mathbf{x} + b$ for all points $(\mathbf{x}, y)$. For example, whatever instruments we use to observe the features 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 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* 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 sum of squared errors. When our prediction for some example i is $\hat{y}^{(i)}$ and the corresponding true label is $y^{(i)}$, the squared error is given by:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{5.1.5}$$

The constant $1/2$ makes no real difference but will prove notationally convenient, cancelling out when we take the dervative 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:

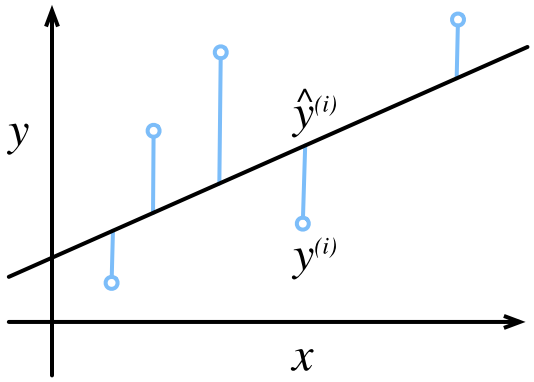


Fig. 5.1.1: Fit data with a linear model.

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, we simply average (or equivalently, sum) the losses on the training set.

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{5.1.6}$$

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

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}}\ L(\mathbf{w}, b). \tag{5.1.7}$$

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, yielding a global optimum.

To start, we can subsume the bias b into the parameter $\mathbf{w}$ by appending a column to the design matrix consisting of all $1s$. Then our prediction problem is to minimize $||\mathbf{y} - X\mathbf{w}||$. Because this expression has a quadratic form, it is convex, and so long as the problem is not degenerate (our features are linearly independent), it is strictly convex.

Thus there is just one critical point on the loss surface and it corresponds to the global minimum. Taking the derivative of the loss with respect to $\mathbf{w}$ and setting it equal to 0 yields the analytic solution:

$$\mathbf{w}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y \tag{5.1.8}$$

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.

Gradient descent

Even in cases where we cannot solve the models analytically, and even when the loss surfaces are high-dimensional and nonconvex, it turns out that we can still train models effectively in practice. Moreover, for many tasks, these 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*. On convex loss surfaces, it will eventually converge to a global minimum, and while the same cannot be said for nonconvex surfaces, it will at least lead towards a (hopefully good) local minimum.

The most naive application of gradient descent consists of taking the derivative of the true loss, 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 computer the update, a variant called *stochastic gradient descent*.

In each iteration, we first randomly sample a minibatch $\mathcal{B}$ consisting of a fixed number of training data examples.
We then compute the derivative (gradient) of the average loss on the mini batch with regard to the model parameters.
Finally, we multiply the gradient by a predetermined step size $\eta > 0$ and subtract the resulting term from the current parameter values.

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

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b) \tag{5.1.9}$$

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 batches from the the data (many times), updating the parameters in the direction of the negative gradient.

For quadratic losses and linear functions, we can write this out explicitly as follows: Note that **w** and **x** are vectors. Here, the more elegant vector notation makes the math much more readable than expressing things in terms of coefficients, say $w_1, w_2, \ldots, w_d$.

$$\begin{aligned} \mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) &= w - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right), \\ b \leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_b l^{(i)}(\mathbf{w}, b) &= b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right). \end{aligned} \tag{5.1.10}$$

In the above equation, $|\mathcal{B}|$ 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 *hyper-parameters*. *Hyperparameter tuning* is the process by which these are chosen, and typically requires that we adjust the hyperparameters based on the results of the inner (training) loop as assessed on a separate *validation* split of the data.

After training for some predetermined number of iterations (or until some other stopping criteria is met), we record the estimated model parameters, denoted $\hat{\mathbf{w}}, \hat{b}$ (in general the "hat" symbol denotes estimates). 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 a local minimum it cannot achieve it exactly in a finite number of steps.

Linear regression happens to be a convex learning problem, and thus there is only one (global) minimum. 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 stuggle to find parameters that minimize the loss *on training data*. 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 $\hat{\mathbf{w}}^\top x + \hat{b}$, we can now estimate the price of a new house (not contained in the training data) given its area x_1 and age (year) x_2. Estimating targets given features is commonly called *prediction* and *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.

Vectorization for Speed

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

To illustrate why this matters so much, we can consider two methods for adding vectors. To start we instantiate two 100000-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 `np`.

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

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 (hereafter accessed via the `d2l` package to track the running time.

```
# Saved in the d2l package for later use
class Timer(object):
    """Record multiple running times."""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        # Start the timer
        self.start_time = time.time()

    def stop(self):
        # Stop the timer and record the time in a list
        self.times.append(time.time() - self.start_time)
        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
```

(continues on next page)

(continued from previous page)

```
        return sum(self.times)

    def cumsum(self):
        # Return the accumuated times
        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.

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

```
'3.87399 sec'
```

Alternatively, we rely on `np` to compute the elementwise sum:

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

```
'0.00032 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 math to the library and need not write as many calculations ourselves, reducing the potential for errors.

5.1.2 The Normal Distribution and Squared Loss

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

Recall from the above that the squared loss $l(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ has many convenient properties. These include a simple derivative $\partial_{\hat{y}} l(y, \hat{y}) = (\hat{y} - y)$.

As we mentioned earlier, linear regression was invented by Gausss 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 is given as follows:

$$p(z) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(z - \mu)^2\right) \tag{5.1.11}$$

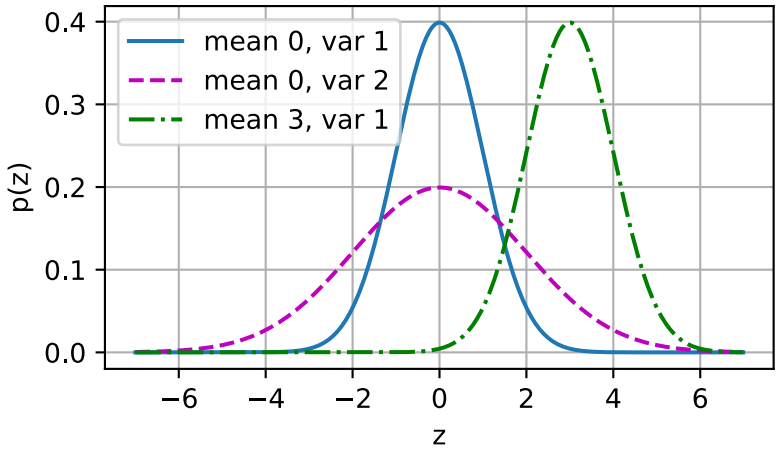
Below we define a Python function to compute the normal distribution.

```
x = np.arange(-7, 7, 0.01)

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

We can now visualize the normal distributions.

```
# Mean and variance pairs
parameters = [(0,1), (0,2), (3,1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in parameters], xlabel='z',
         ylabel='p(z)', figsize=(4.5, 2.5),
         legend=['mean %d, var %d' % (mu, sigma) for mu, sigma in parameters])
```



As you 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 is to formally assume that observations arise from noisy observations, where the noise is normally distributed as follows

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{5.1.12}$$

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

$$p(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right) \tag{5.1.13}$$

Now, according to the *maximum likelihood principle*, the best values of b and $\mathbf{w}$ are those that maximize the *likelihood* of the entire dataset:

$$P(Y \mid X) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}) \tag{5.1.14}$$

Estimators chosen according to the *maximum likelihood principle* are called *Maximum Likelihood Estimators* (MLE). 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 (NLL)* $-\log p(\mathbf{y}|\mathbf{X})$. Working out the math gives us:

$$-\log p(\mathbf{y}|\mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2 \tag{5.1.15}$$

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 objective introduced earlier, but for the multiplicative constant $\frac{1}{\sigma^2}$. Fortunately, the solution does not depend on σ. It follows that minimizing squared error is equvalent to maximum likelihood estimation of a linear model under the assumption of additive Gaussian noise.

5.1.3 From Linear Regression to Deep Networks

So far we only talked about linear functions. 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 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. 5.1.2, we depict our linear model as a neural network. Note that these diagrams indicate the connectivity pattern (here, each input is connected to the output) but not the values taken by the weights or biases.

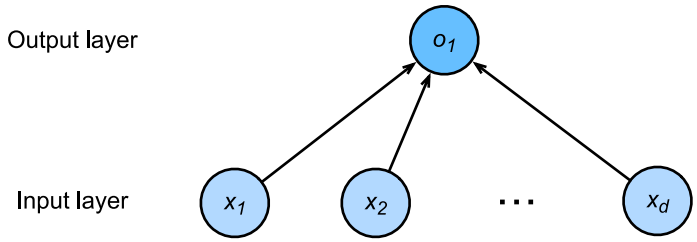


Fig. 5.1.2: Linear regression is a single-layer neural network.

Because there is just a single computed neuron (node) in the graph (the input values are not computed but given), we can think of linear models as neural networks consisting of just a single artificial neuron. Since for this model, every input is connected to every output (in this case there is only one output!), we can regard this transformation as a *fully-connected layer*, also commonly called a *dense layer*. We will talk a lot more about networks composed of such layers in the next chapter on multilayer perceptrons.

Biology

Although linear regression (invented in 1795) predates computational neuroscience, so 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 looked when they began to develop models of artificial neurons, consider the following cartoonish picture of a biological neuron, 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*.

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 along could express owes to our study of real biological neural systems.

At the same time, most reserach 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[50], pointed out that although airplanes might have been *inspired* by birds, ornithology has not been the primary driver of aeronautics

[50] https://en.wikipedia.org/wiki/Artificial_Intelligence:_A_Modern_Approach

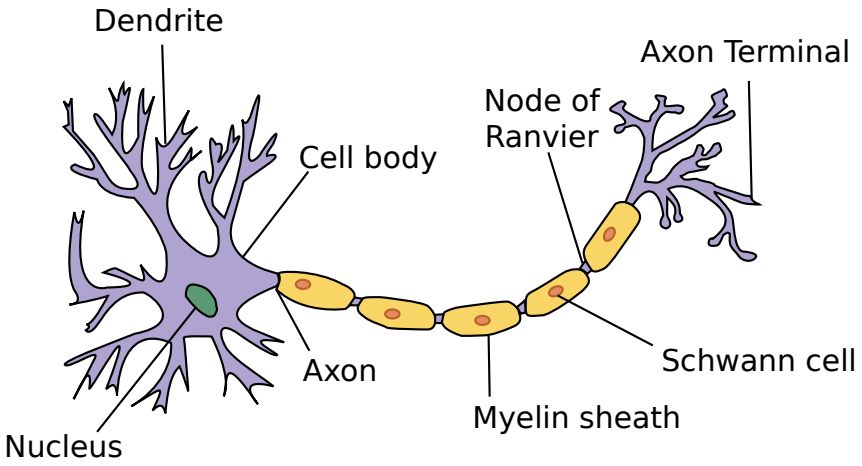


Fig. 5.1.3: The real neuron

innovaton for some centuries. Likewise, inspiration in deep learning these days comes in equal or greater measure from mathematics, statistics, and computer science.

5.1.4 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 can mean the same thing.
- Linear models are neural networks, too.

5.1.5 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.
    - Find a closed-form solution for the optimal value of b.
    - How does this problem and its solution relate to the normal distribution?
2. Derive the closed-form 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 X consisting of all ones).
    - Write out the optimization problem in matrix and vector notation (treat all the data as a single matrix, all the target values as a single vector).
    - Compute the gradient of the loss with respect to w.
    - Find the closed form solution by setting the gradient equal to zero and solving the matrix equation.
    - 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, $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$.
    - Write out the negative log-likelihood of the data under the model $-\log P(Y \mid X)$.

- Can you find a closed form solution?
- 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?

5.1.6 Scan the QR Code to Discuss[51]

5.2 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 gradient descent optimizer. While modern deep learning frameworks can automate nearly all of this work, implementing things from scratch is the only to make sure that you really know what you are doing. Moreover, when it comes time to customize models, defining our own layers, loss functions, etc., understanding how things work under the hood will prove handy. In this section, we will rely only on `ndarray` and `autograd`. Afterwards, we will introduce a more compact implementation, taking advantage of Gluon's bells and whistles. To start off, we import the few required packages.

```
%matplotlib inline
import d2l
from mxnet import autograd, np, npx
import random
npx.set_np()
```

5.2.1 Generating Data Sets

To keep things simple, we will construct an artificial dataset according to a linear model with additive noise. Out 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 generated a datset containing 1000 examples, each consisting of 2 features sampled from a standard normal distribution. Thus our synthetic dataset will be an object $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$.

The true parameters generating our data 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 noise term ϵ:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon \tag{5.2.1}$$

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:

[51] https://discuss.mxnet.io/t/2331

```
# Saved in the d2l package for later use
def synthetic_data(w, b, num_examples):
    """generate y = X w + 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

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

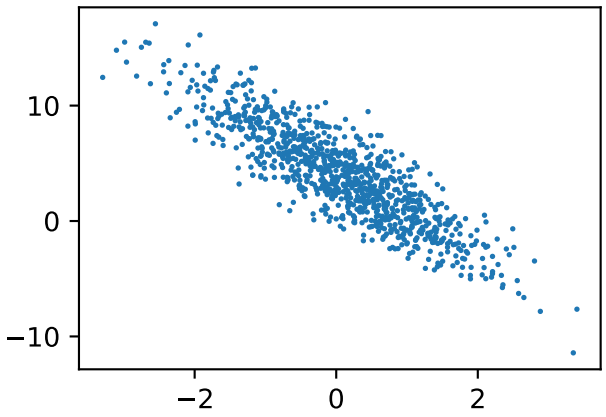
Note that each row in `features` consists of a 2-dimensional data point and that each row in `labels` consists of a 1-dimensional target value (a scalar).

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

```
features: [2.2122064 0.7740038]
label: 6.000587
```

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

```
d2l.set_figsize((3.5, 2.5))
d2l.plt.scatter(features[:, 1].asnumpy(), labels.asnumpy(), 1);
```



5.2.2 Reading Data

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 algortihms, its worth defining a utility function to shuffle the data and access it in minibatches.

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

```
[[-1.2382596  -0.05393763]
 [-2.825597   -1.3722466 ]
 [ 1.0955427   0.7685205 ]
 [ 1.0132014   0.9695031 ]
 [ 0.20988831 -1.0934474 ]
 [ 1.3551259  -0.7086181 ]
 [-0.27408606  0.2229926 ]
 [-2.7660658  -0.6561834 ]
 [ 1.063981    1.4385059 ]
 [-0.04226796 -0.14354235]]
 [1.9238918  3.2079732  3.7752948  2.954539   8.336128   9.321416
 2.8890834  0.89435107 1.4290493  4.608826  ]
```

As we run the iterator, we obtain distinct minibatches successively until all the data has been exhausted (try this). While the iterator 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 data in memory and that we perform lots of random memory access. The built-in iterators implemented in Apache MXNet are considerably efficient and they can deal both with data stored on file and data fed via a data stream.

5.2.3 Initialize Model Parameters

Before we can begin optimizing our model's parameters by 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, setting the bias b to 0.

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

Now that we have initialized our parameters, our next task is to update them until they fit our data sufficiently well. Each update requires taking the gradient (a multi-dimensional derivative) of our loss function with respect to the parameters.

Given this gradient, we can update each parameter in the direction that reduces the loss.

Since nobody wants to compute gradients explicitly (this is tedious and error prone), we use automatic differentiation to compute the gradient. See Section 4.6 for more details. Recall from the autograd chapter that in order for `autograd` to know that it should store a gradient for our parameters, we need to invoke the `attach_grad` function, allocating memory to store the gradients that we plan to take.

```
w.attach_grad()
b.attach_grad()
```

5.2.4 Define 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 examples $\mathbf{X}$ and the models weights w, and add the offset b to each example. Note that below `np.dot(X, w)` is a vector and `b` is a scalar. Recall that when we add a vector and a scalar, the scalar is added to each component of the vector.

```
# Saved in the d2l package for later use
def linreg(X, w, b):
    return np.dot(X, w) + b
```

5.2.5 Define 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 the previous section. 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 be the same as the `y_hat` shape.

```
# Saved in the d2l package for later use
def squared_loss(y_hat, y):
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

5.2.6 Define the Optimization Algorithm

As we discussed in the previous section, 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 stochastic gradient descent (SGD).

At each step, using one batch randomly drawn from our dataset, we will estimate the gradient of the loss with respect to our parameters. Next, we will update our parameters (a small amount) in the direction that reduces the loss. Recall from Section 4.6 that after we call `backward` each parameter (`param`) will have its gradient stored in `param.grad`. The following code applies the SGD 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 batch 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.

```
# Saved in the d2l package for later use
def sgd(params, lr, batch_size):
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

5.2.7 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 minibatches of models, first passing them through our model to obtain a set of predictions. After calculating the loss, we call the `backward` function to initiate the backwards pass through the network, storing the gradients with respect to each parameter in its corresponding `.grad` attribute. Finally, we will call the optimization algorithm `sgd` to update the model parameters. Since we previously set the batch size `batch_size` to 10, the loss shape `l` for each minibatch is (10, 1).

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 the code below, `l` is a vector of the losses for each example in the minibatch. Because `l` is not a scalar variable, running `l.backward()` adds together the elements in `l` to obtain the new variable and then calculates the gradient.

In each epoch (a pass through the data), we will iterate through the entire dataset (using the `data_iter` function) once passing through every examples in the training dataset (assuming the number of examples is divisible by the batch size). The number of epochs `num_epochs` and the learning rate `lr` are both hyper-parameters, which we set here to 3 and 0.03, respectively. Unfortunately, setting hyper-parameters is tricky and requires some adjustment by trial and error. We elide these details for now but revise them later in Section 12.

```
lr = 0.03  # Learning rate
num_epochs = 3  # Number of iterations
net = linreg  # Our fancy linear model
loss = squared_loss  # 0.5 (y-y')^2

for epoch in range(num_epochs):
    # Assuming the number of examples can be divided by the batch size, all
    # the examples in the training data set are used once in one epoch
    # iteration. The features and tags of minibatch examples are given by X
    # and y respectively
    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
        l.backward()  # Compute gradient on l with respect to [w,b]
        sgd([w, b], lr, batch_size)  # Update parameters using their gradient
    train_l = loss(net(features, w, b), labels)
    print('epoch %d, loss %f' % (epoch + 1, train_l.mean().asnumpy()))
```

```
epoch 1, loss 0.040704
epoch 2, loss 0.000156
epoch 3, loss 0.000050
```

In this case, because we synthesized the data 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('Error in estimating w', true_w - w.reshape(true_w.shape))
print('Error in estimating b', true_b - b)
```

```
Error in estimating w [6.3574314e-04 5.1259995e-05]
Error in estimating b [0.00029945]
```

Note that we should not take it for granted that we are able to recover the parameters accurately. This only happens for a special category problems: strongly convex optimization problems with 'enough' data to ensure that the noisy samples allow us to recover the underlying dependency. In most cases this is *not* the case. In fact, the parameters of a deep network are rarely the same (or even close) between two different runs, unless all conditions are identical, including the order in which the data is traversed. However, in machine learning, we are typically less concerned with recovering true underlying parameters, and more concerned with parameters that lead to 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 accurate prediction.

5.2.8 Summary

We saw how a deep network can be implemented and optimized from scratch, using just `ndarray` and `autograd`, without any need for defining layers, fancy optimizers, etc. This 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.

5.2.9 Exercises

1. What would happen if we were to initialize the weights $\mathbf{w} = 0$. Would the algorithm still work?
2. Assume that you are Georg Simon Ohm[52] trying to come up with a model between voltage and current. Can you use `autograd` to learn the parameters of your model.
3. Can you use Planck's Law[53] to determine the temperature of an object using spectral energy density?
4. What are the problems you might encounter if you wanted to extend `autograd` to 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?

5.2.10 Scan the QR Code to Discuss[54]

[52] https://en.wikipedia.org/wiki/Georg_Ohm
[53] https://en.wikipedia.org/wiki/Planck%27s_law
[54] https://discuss.mxnet.io/t/2332

5.3 Concise Implementation of Linear Regression

Broad and intense interest in deep learning for the past several years has inspired both 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 the previous section, we relied only on (i) `ndarray` for data storage and linear algebra; and (ii) `autograd` for calculating derivatives. In practice, because data iterators, loss functions, optimizers, and neural network layers (and some whole architectures) 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 5.2 concisely by using Gluon.

5.3.1 Generating Data Sets

To start, we will generate the same dataset as in the previous section.

```
import d2l
from mxnet import autograd, np, npx, gluon
npx.set_np()

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

5.3.2 Reading Data

Rather than rolling our own iterator, we can call upon Gluon's `data` module to read data. The first step will be to instantiate an `ArrayDataset`. This object's constructor takes one or more `ndarrays` as arguments. Here, we pass in `features` and `labels` as arguments. Next, we will use the ArrayDataset to instantiate a DataLoader, which also requires that we specify a `batch_size` and specify a Boolean value `shuffle` indicating whether or not we want the `DataLoader` to shuffle the data on each epoch (pass through the dataset).

```
# Saved in the d2l package for later use
def load_array(data_arrays, batch_size, is_train=True):
    """Construct a Gluon data loader"""
    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 the previous section. To verify that it is working, we can read and print the first minibatch of instances.

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

```
[[ 0.82885706  0.5436285 ]
 [ 0.8269238   1.0562588 ]
 [ 0.04628786  0.48509744]
 [-1.8385648   0.0952221 ]
 [-0.5227964  -0.649588  ]
```

(continues on next page)

(continued from previous page)

```
[ 1.0708472  -2.6909585 ]
[ 0.29560572 -1.2169206 ]
[ 0.27814385  0.09634511]
[-0.6434393   0.39862037]
[-2.0710864  -0.08075491]]
[ 4.0100265   2.2603266   2.6319473   0.20533901  5.3611174  15.49145
  8.941942    4.4285975   1.5484271   0.34214625]
```

5.3.3 Define the Model

When we implemented linear regression from scratch (in :numref“sec_linear_scratch“), 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 weel.

For standard operations, we can use Gluon'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. To define a linear model, we first import the `nn` module, which defines a large number of neural network layers (note that "nn" is an abbreviation for neural networks). We will first define a model variable `net`, which will refer to an instance of the `Sequential` class. In Gluon, `Sequential` defines a container for several layers that will be chained together. Given input data, a `Sequential` 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.

```
from mxnet.gluon import nn
net = nn.Sequential()
```

Recall the architecture of a single-layer network. The layer is said to be *fully-connected* because each of its inputs are 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.

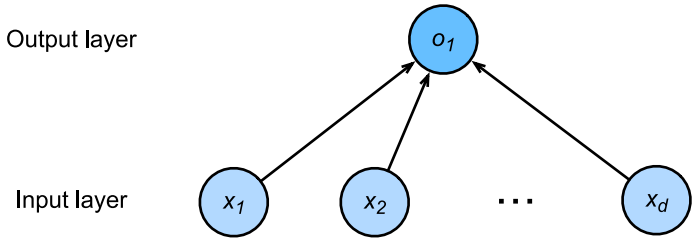


Fig. 5.3.1: Linear regression is a single-layer neural network.

```
net.add(nn.Dense(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 in the chapter "Deep Learning Computation".

5.3.4 Initialize Model Parameters

Before using `net`, we need to initialize the model parameters, such as the weights and biases in the linear regression model. 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. By calling `init.Normal(sigma=0.01)`, 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 by default. Both the weight vector and bias will have attached gradients.

```
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 scenes, 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.

5.3.5 Define the Loss Function

In Gluon, the `loss` module defines various loss functions. We will the imported module `loss` with the pseudonym `gloss`, to avoid confusing it for the variable holding our chosen loss function. In this example, we will use the Gluon implementation of squared loss (`L2Loss`).

```
from mxnet.gluon import loss as gloss
loss = gloss.L2Loss()  # The squared loss is also known as the L2 norm loss
```

5.3.6 Define the Optimization Algorithm

Minibatch SGD and related variants are standard tools for optimizing neural networks and thus `Gluon` supports SGD alongside a number of variations on this algorithm through its `Trainer` class. When we instantiate the `Trainer`, we will specify the parameters to optimize over (obtainable from our net via `net.collect_params()`), the optimization algortihm we wish to use (`sgd`), and a dictionary of hyper-parameters required by our optimization algorithm. SGD just requires that we set the value `learning_rate`, (here we set it to 0.03).

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

5.3.7 Training

You might have noticed that expressing our model through Gluon requires comparatively few lines of code. We did not have to individually allocate parameters, define our loss function, or implement stochastic gradient descent. Once we start working with much more complex models, Gluon's advantages 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 pass).
- Calculate gradients by calling `l.backward()` (the backward pass).
- Update the model parameters by invoking our SGD optimizer (note that `trainer` already knows which parameters to optimize over, so we just need to pass in the minibatch size.

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

```
num_epochs = 3
for epoch in range(1, num_epochs + 1):
    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('epoch %d, loss: %f' % (epoch, l.mean().asnumpy()))
```

```
epoch 1, loss: 0.040636
epoch 2, loss: 0.000155
epoch 3, loss: 0.000050
```

Below, we compare the model parameters learned by training on finite data and the actual parameters that generated our dataset. To access parameters with Gluon, we first access the layer that we need from `net` and then access that layer's weight (`weight`) and bias (`bias`). To access each parameter's values as an `ndarray`, we invoke its `data` method. As in our from-scratch implementation, note that our estimated parameters are close to their ground truth counterparts.

```
w = net[0].weight.data()
print('Error in estimating w', true_w.reshape(w.shape) - w)
b = net[0].bias.data()
print('Error in estimating b', true_b - b)
```

```
Error in estimating w [[ 0.00018573 -0.00039506]]
Error in estimating b [0.00039959]
```

5.3.8 Summary

- Using Gluon, we can implement models much more succinctly.
- 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).

5.3.9 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`?

5.3.10 Scan the QR Code to Discuss[55]

5.4 Softmax Regression

In Section 5.1, we introduced linear regression, working through implementations from scratch in Section 5.2 and again using Gluon in Section 5.3 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 (the *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 likley *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; 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.

5.4.1 Classification Problems

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 {dog, cat, 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 {baby, toddler, adolescent, young adult, adult, 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.

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\} \tag{5.4.1}$$

[55] https://discuss.mxnet.io/t/2333

In our case, 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".

Network Architecture

In order to estimate the conditional probabilities associated with each classes, we need a model with multiple outputs, one per class. To address classification with linear models, we will need as many linear functions as we have outputs. Each output will correpsond to its own linear 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:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned} \tag{5.4.2}$$

We can depict this calculation with the neural network diagram below. 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.

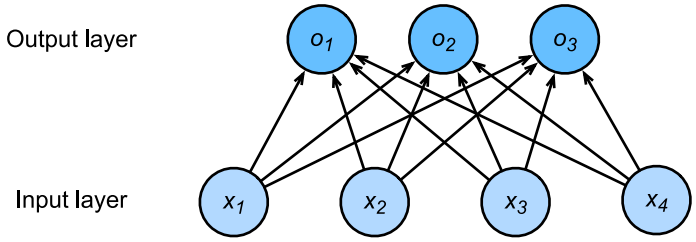


Fig. 5.4.1: Softmax regression is a single-layer neural network.

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 a given example $\mathbf{x}$, our outputs are given by a matrix-vector product of our weights by our inputs plus our biases $\mathbf{b}$.

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 *argmax* of the predicted probabilities.

Put formally, we would like outputs $\hat{y}_k$ that we can interpret as the probability that a given item belongs to class k. Then we can choose the class with the largest output value as our prediction $\text{argmax}_k\, y_k$. For example, if $\hat{y}_1$, $\hat{y}_2$, and $\hat{y}_3$ are 0.1, $.8$, 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. Nothing constrains these numbers to sum to 1. Moreover, depending on the inputs, they can take negative values. These violate basic axioms of probability presented in Section 4.7

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 $.5$, 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).

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)} \tag{5.4.3}$$

It is easy to see $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ with $0 \leq \hat{y}_i \leq 1$ for all i. Thus, $\hat{y}$ is a proper probability distribution and the values of $\hat{\mathbf{y}}$ can be interpreted accordingly. Note that the softmax operation does not change the ordering among the logits, and thus we can still pick out the most likely class by:

$$\hat{\imath}(\mathbf{o}) = \operatorname*{argmax}_i o_i = \operatorname*{argmax}_i \hat{y}_i \tag{5.4.4}$$

The logits $\mathbf{o}$ then are simply the pre-softmax values that determining the probabilities assigned to each category. Summarizing it all in vector notation we get $\mathbf{o}^{(i)} = \mathbf{W}\mathbf{x}^{(i)} + \mathbf{b}$, where $\hat{\mathbf{y}}^{(i)} = \mathrm{softmax}(\mathbf{o}^{(i)})$.

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 dimensionality d and batch size n. Moreover, assume that we have q categories (outputs). 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 $\mathbf{b} \in \mathbb{R}^q$.

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b} \\ \hat{\mathbf{Y}} &= \mathrm{softmax}(\mathbf{O}) \end{aligned} \tag{5.4.5}$$

This accelerates the dominant operation into a matrix-matrix product $\mathbf{W}\mathbf{X}$ vs the matrix-vector products we would be executing if we processed one example at a time. The softmax itself can be computed by exponentiating all entries in $\mathbf{O}$ and then normalizing them by the sum.

5.4.2 Loss Function

Next, we need a *loss function* to measure the quality of our predicted probabilities. We will rely on *likelihood maximization*, the very same concept that we encountered when providing a probabilistic justification for the least squares objective in linear regression (Section 5.1).

Log-Likelihood

The softmax function gives us a vector $\hat{\mathbf{y}}$, which we can interpret as estimated conditional probabilities of each class given the input x, e.g., $\hat{y}_1 = \hat{P}(y = \text{cat} \mid \mathbf{x})$. We can compare the estimates with reality by checking how probable the *actual* classes are according to our model, given the features.

$$P(Y \mid X) = \prod_{i=1}^{n} P(y^{(i)} \mid x^{(i)}) \text{ and thus } -\log P(Y \mid X) = \sum_{i=1}^{n} -\log P(y^{(i)} \mid x^{(i)}) \tag{5.4.6}$$

Maximizing $P(Y \mid X)$ (and thus equivalently minimizing $-\log P(Y \mid X)$) corresponds to predicting the label well. This yields the loss function (we dropped the superscript (i) to avoid notation clutter):

$$l = -\log P(y \mid x) = -\sum_j y_j \log \hat{y}_j \tag{5.4.7}$$

For reasons explained later on, this loss function is commonly called the *cross-entropy* loss. Here, we used that by construction $\hat{y}$ is a discrete probability distribution and that the vector $\mathbf{y}$ is a one-hot vector. Hence the the sum over all

coordinates j vanishes for all but one term. Since all $\hat{y}_j$ are probabilities, their logarithm is never larger than 0. Consequently, the loss function cannot be minimized any further if we correctly predict y with *certainty*, i.e., if $P(y \mid x) = 1$ for the correct label. Note that this is often not possible. 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 while understanding a bit better how it is computed. Plugging o into the definition of the loss l and using the definition of the softmax we obtain:

$$l = -\sum_j y_j \log \hat{y}_j = \sum_j y_j \log \sum_k \exp(o_k) - \sum_j y_j o_j = \log \sum_k \exp(o_k) - \sum_j y_j o_j \tag{5.4.8}$$

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

$$\partial_{o_j} l = \frac{\exp(o_j)}{\sum_k \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j = P(y = j \mid x) - y_j \tag{5.4.9}$$

In other words, the gradient is the difference between the probability assigned to the true class by our model, as expressed by the probability $P(y \mid x)$, and what actually happened, as expressed by y. 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[56] 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 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 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.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_j y_j \log \hat{y}_j \tag{5.4.10}$$

This loss is called the cross-entropy loss and it is one of the most commonly used losses for multiclass classification. We can demystify the name by introducing the basics of information theory.

5.4.3 Information Theory Basics

Information theory deals with the problem of encoding, decoding, transmitting and manipulating information (aka 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[57] of a distribution p, and it is captured by the following equation:

$$H[p] = \sum_j -p(j) \log p(j) \tag{5.4.11}$$

[56] https://en.wikipedia.org/wiki/Exponential_family
[57] https://en.wikipedia.org/wiki/Entropy

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. One nat is $\frac{1}{\log(2)} \approx 1.44$ bit. $H[p]/2$ is often also called the binary entropy.

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 is it boring, but it is 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 some times be surprised. Our surpise is greater when we assigned an event lower probability. For reasons that we will elaborate in the appendix, Claude Shannon settled on $\log(1/p(j)) = -\log p(j)$ to quantify one's *surprisal* at observing an event j having assigned it a (subjective) probability $p(j)$. The entropy is then the *expected surprisal* when one assigned the correct probabilities (that truly match the data-generating process). The entropy of the data is then the least surprised that one can ever be (in expectation).

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 :math:'p' to :math:'q'*, denoted H(p, q), is the expected surprisal of an observer with subjective probabilities q upon seeing data that was actually generated according to probabilities p. The lowest possible cross-entropy is achieved when $p = q$. In this case, the cross-entropy from p to q is $H(p, p) = H(p)$. Relating this back to our classification objective, even if we get the best possible predictions, if the best possible possible, then we will never be perfect. Our loss is lower-bounded by the entropy given by the actual conditional distributions $P(\mathbf{y} \mid \mathbf{x})$.

Kullback Leibler Divergence

Perhaps the most common way to measure the distance between two distributions is to calculate the Kullback Leibler divergence $D(p\|q)$. This is simply the difference between the cross-entropy and the entropy, i.e., the additional cross-entropy incurred over the irreducible minimum value it could take:

$$D(p\|q) = H(p, q) - H[p] = \sum_j p(j) \log \frac{p(j)}{q(j)} \tag{5.4.12}$$

$$Notethatinclassificatio, wedonotknowthetrue : math : `p`, sowe$$

cannot compute the entropy directly. However, because the entropy is out of our control, minimizing $D(p\|q)$ with respect to q is equivalent to minimizing the cross-entropy loss.

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 surprise (and thus the number of bits) required to communicate the labels.

5.4.4 Model Prediction and Evaluation

After training the softmax regression model, given any example features, we can predict the probability of each output category. Normally, we use the category with the highest predicted probability as the output category. The prediction is correct if it is consistent with the actual category (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 a nd the total number of predictions.

5.4.5 Summary

- We introduced the softmax operation which takes a vector maps it into probabilities.
- Softmax regression applies to classification problems. It uses the probability distribution of the output category 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.

5.4.6 Exercises

1. Show that the Kullback-Leibler divergence $D(p\|q)$ is nonnegative for all distributions p and q. Hint - use Jensen's inequality, i.e., use the fact that $-\log x$ is a convex function.
2. Show that $\log \sum_j \exp(o_j)$ is a convex function in o.
3. We can explore the connection between exponential families and the softmax in some more depth
   - Compute the second derivative of the cross-entropy loss $l(y, \hat{y})$ for the softmax.
   - Compute the variance of the distribution given by $\text{softmax}(o)$ and show that it matches the second derivative computed above.
4. Assume that we three classes which occur with equal probability, i.e., the probability vector is $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
   - What is the problem if we try to design a binary code for it? Can we match the entropy lower bound on the number of bits?
   - Can you design a better code. Hint - what happens if we try to encode two independent observations? What if we encode n observations jointly?
5. Softmax is a misnomer for the mapping introduced above (but everyone in deep learning uses it). The real softmax is defined as $\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$.
   - Prove that $\text{RealSoftMax}(a, b) > \max(a, b)$.
   - Prove that this holds for $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b)$, provided that $\lambda > 0$.
   - Show that for $\lambda \to \infty$ we have $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$.
   - What does the soft-min look like?
   - Extend this to more than two numbers.

5.4.7 Scan the QR Code to Discuss[58]

[58] https://discuss.mxnet.io/t/2334

5.5 Image Classification Data (Fashion-MNIST)

In Section 18.9, we trained a naive Bayes classifier, using the MNIST dataset introduced in 1998 (LeCun et al., 1998). While MNIST 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 d2l
from mxnet import gluon
import sys
d2l.use_svg_display()
```

5.5.1 Getting the data

Just as with MNIST, Gluon makes it easy to download and load the FashionMNIST dataset into memory via the `FashionMNIST` class contained in `gluon.data.vision`. We briefly work through the mechnics of loading and exploring the dataset below. Please refer to Section 18.9 for more details on loading data.

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

FashionMNIST conists of images from 10 categories, each represented by 6k images in the training set and by 1k in the test set. Consequently the training set and the test set contain 60k and 10k images, respectively.

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

```
(60000, 10000)
```

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.

```
# Saved in the d2l package for later use
def get_fashion_mnist_labels(labels):
    text_labels = ['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.

```
# Saved in the d2l package for later use
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
    """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)
```

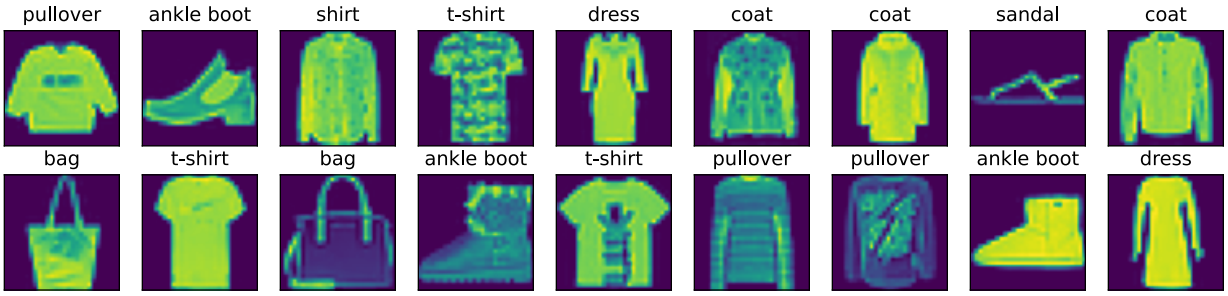
(continues on next page)

(continued from previous page)

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



5.5.2 Reading a Minibatch

To make our life easier when reading from the training and test sets, we use a `DataLoader` rather than creating one from scratch, as we did in Section 5.2. Recall that at each iteration, a `DataLoader` reads a minibatch of data with size `batch_size` each time.

During training, reading data can be a significant performance bottleneck, especially when our model is simple or when our computer is fast. A handy feature of Gluon's `DataLoader` is the ability to use multiple processes to speed up data reading. For instance, we can set aside 4 processes to read the data (via `num_workers`). Because this feature is not currently supported on Windows the following code checks the platform to make sure that we do not saddle our Windows-using friends with error messages later on.

```
# Saved in the d2l package for later use
def get_dataloader_workers(num_workers=4):
    # 0 means no additional process is used to speed up the reading of data.
    if sys.platform.startswith('win'):
        return 0
    else:
        return num_workers
```

Below, we convert the image data from uint8 to 32-bit floating point numbers using the `ToTensor` class. Additionally, the transformer will divide all numbers by 255 so that all pixels have values between 0 and 1. The `ToTensor` class also moves the image channel from the last dimension to the first dimension to facilitate the convolutional neural network calculations introduced later. Through the `transform_first` function of the dataset, we apply the transformation of `ToTensor` to the first element of each instance (image and label).

```
batch_size = 256
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
'%.2f sec' % timer.stop()
```

```
'1.48 sec'
```

5.5.3 Put 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.

```
# Saved in the d2l package for later use
def load_data_fashion_mnist(batch_size, resize=None):
    """Download the Fashion-MNIST dataset and then load into memory."""
    dataset = gluon.data.vision
    trans = [dataset.transforms.Resize(resize)] if resize else []
    trans.append(dataset.transforms.ToTensor())
    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 verify that image resizing works.

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

```
(32, 1, 64, 64)
```

We are now ready to work with the FashionMNIST dataset in the sections that folow.

5.5.4 Summary

- Fashion-MNIST is an apparel classification dataset consisting of images representing 10 categories.
- We will use this datasrt in subsequent sections and chapters to evaluate various classification algorithms.
- We store the shape of each 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 iterators that exploit multi-threading to avoid slowing down your training loop.

5.5.5 Exercises

1. Does reducing the `batch_size` (for instance, to 1) affect read performance?
2. For non-Windows users, try modifying `num_workers` to see how it affects read performance. Plot the performance against the number of works employed.
3. Use the MXNet documentation to see which other datasets are available in `mxnet.gluon.data.vision`.
4. Use the MXNet documentation to see which other transformations are available in `mxnet.gluon.data.vision.transforms`.

5.5.6 Scan the QR Code to Discuss[59]

5.6 Implementation of Softmax Regression from Scratch

Just as we implemented linear regression from scratch, we believe that multiclass logistic (softmax) regression is similarly fundamental and you ought to know the gory details of how to implement it yourself. As with linear regression, after doing things by hand we will breeze through an implementation in Gluon for comparison. To begin, let's import the familiar packages.

```
import d2l
from mxnet import autograd, np, npx, gluon
from IPython import display
npx.set_np()
```

We will work with the Fashion-MNIST dataset, just introduced in Section 5.5, setting up an iterator with batch size 256.

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

5.6.1 Initialize Model Parameters

As in our linear regression example, each example here will be represented by a fixed-length vector. Each example in the raw data is a 28×28 image. In this chapter, we will flatten each image, treating them as 784 1D vectors. 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 categories. Because our dataset has 10 categories, our network will have an output dimension of 10. Consequently, our weights will constitute a 784×10 matrix and the biases will constitute a 1×10 vector. As with linear regression, we will initialize our weights W with Gaussian noise and our biases to take the initial value 0.

[59] https://discuss.mxnet.io/t/2335

```
num_inputs = 784
num_outputs = 10

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

Recall that we need to *attach gradients* to the model parameters. More literally, we are allocating memory for future gradients to be stored and notifiying MXNet that we will want to calculate gradients with respect to these parameters in the future.

```
W.attach_grad()
b.attach_grad()
```

5.6.2 The Softmax

Before implementing the softmax regression model, let's briefly review how operators such as `sum` work along specific dimensions in an `ndarray`. Given a matrix `X` we can sum over all elements (default) or only over elements in the same axis, *i.e.*, the column (`axis=0`) or the same row (`axis=1`). Note that if `X` is an array with shape `(2, 3)` and we sum over the columns (`X.sum(axis=0)`, the result will be a (1D) vector with shape `(3,)`. If we want to keep the number of axes in the original array (resulting in a 2D array with shape `(1,3)`), rather than collapsing out the dimension that we summed over we can specify `keepdims=True` when invoking `sum`.

```
X = np.array([[1, 2, 3], [4, 5, 6]])
print(X.sum(axis=0, keepdims=True), '\n', X.sum(axis=1, keepdims=True))
```

```
[[5. 7. 9.]]
 [[ 6.]
 [15.]]
```

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

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})} \tag{5.6.1}$$

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[60] where a related equation models the distribution over an ensemble of particles).

```
def softmax(X):
    X_exp = np.exp(X)
    partition = X_exp.sum(axis=1, keepdims=True)
    return X_exp / partition  # The broadcast 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. Note that while this looks correct mathematically, we were a bit sloppy in our implementation because failed to take precautions against numerical overflow or underflow due to large (or very small) elements of the matrix, as we did in Section 18.9.

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

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

```
(array([[0.21324193, 0.33961776, 0.1239742 , 0.27106097, 0.05210521],
        [0.11462264, 0.3461234 , 0.19401033, 0.29583326, 0.04941036]]),
 array([1.0000001, 1.       ]))
```

5.6.3 The Model

Now that we have defined the softmax operation, we can implement the softmax regression model. The below code defines the forward pass through the network. Note that we flatten each original image in the batch into a vector with length `num_inputs` with the `reshape` function before passing the data through our model.

```
def net(X):
    return softmax(np.dot(X.reshape(-1, num_inputs), W) + b)
```

5.6.4 The Loss Function

Next, we need to implement the cross-entropy loss function, introduced in Section 5.4. 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 $-\log P(y \mid x)$. Rather than iterating over the predictions with a Python `for` loop (which tends to be inefficient), we can use the `pick` function which allows us to easily select the appropriate terms from the matrix of softmax entries. Below, we illustrate the `pick` function on a toy example, with 3 categories and 2 examples.

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

```
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])
```

5.6.5 Classification Accuracy

Given the predicted probability distribution `y_hat`, we typically choose the class with highest predicted probability whenever we must output a *hard* prediction. Indeed, many applications require that we make a choice. Gmail must catetegorize 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 categories.

When predictions are consistent with the actual category `y`, they are correct. The classification accuracy is the fraction of all predictions that are correct. Although it can be difficult optimize accuracy directly (it is not differentiable), it is often the performance metric that we care most about, and we will nearly always report it when training classifiers.

To compute accuracy we do the following: First, we execute `y_hat.argmax(axis=1)` to gather the predicted classes (given by the indices for the largest entires each row). The result has the same shape as the variable `y`. Now we just need to

check how frequently the two match. Since the equality operator == is datatype-sensitive (e.g., an `int` and a `float32` are never equal), we also need to convert both to the same type (we pick `float32`). The result is an `ndarray` containing entries of 0 (false) and 1 (true). Taking the mean yields the desired result.

```
# Saved in the d2l package for later use
def accuracy(y_hat, y):
    if y_hat.shape[1] > 1:
        return float((y_hat.argmax(axis=1) == y.astype('float32')).sum())
    else:
        return float((y_hat.astype('int32') == y.astype('int32')).sum())
```

We will continue to use the variables `y_hat` and `y` defined in the `pick` function, as the predicted probability distribution and label, respectively. We can see that the first example's prediction category is 2 (the largest element of the row is 0.6 with an index of 2), which is inconsistent with the actual label, 0. The second example's prediction category is 2 (the largest element of the row is 0.5 with an index of 2), which is consistent with the actual label, 2. Therefore, the classification accuracy rate for these two examples is 0.5.

```
y = np.array([0, 2])
accuracy(y_hat, y) / len(y)
```

```
0.5
```

Similarly, we can evaluate the accuracy for model `net` on the dataset (accessed via `data_iter`).

```
# Saved in the d2l package for later use
def evaluate_accuracy(net, data_iter):
    metric = Accumulator(2) # num_corrected_examples, num_examples
    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 accumulated sum over multiple numbers.

```
# Saved in the d2l package for later use
class Accumulator(object):
    """Sum a list of numbers over time"""
    def __init__(self, n):
        self.data = [0.0] * n
    def add(self, *args):
        self.data = [a+b for a, b in zip(self.data, args)]
    def reset(self):
        self.data = [0] * len(self.data)
    def __getitem__(self, i):
        return self.data[i]
```

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)
```

```
0.0925
```

5.6.6 Model Training

The training loop for softmax regression should look strikingly familiar if you read through our implementation of linear regression in Section 5.2. Here we refactor the implementation to make it reusable. First, we define a function to train for one data epoch. Note that `updater` is general function to update the model parameters, which accepts the batch size as an argument. It can be either a wrapper of `d2l.sgd` or a Gluon trainer.

```
# Saved in the d2l package for later use
def train_epoch_ch3(net, train_iter, loss, updater):
    metric = Accumulator(3) # train_loss_sum, train_acc_sum, num_examples
    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 draw data in animation. Again, it aims to simplify the codes in later chapters.

```
# Saved in the d2l package for later use
class Animator(object):
    def __init__(self, xlabel=None, ylabel=None, legend=[], xlim=None,
                 ylim=None, xscale='linear', yscale='linear', fmts=None,
                 nrows=1, ncols=1, figsize=(3.5, 2.5)):
        """Incrementally plot multiple lines."""
        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 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)]
        if not self.fmts: self.fmts = ['-'] * 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)
```

(continues on next page)

(continued from previous page)

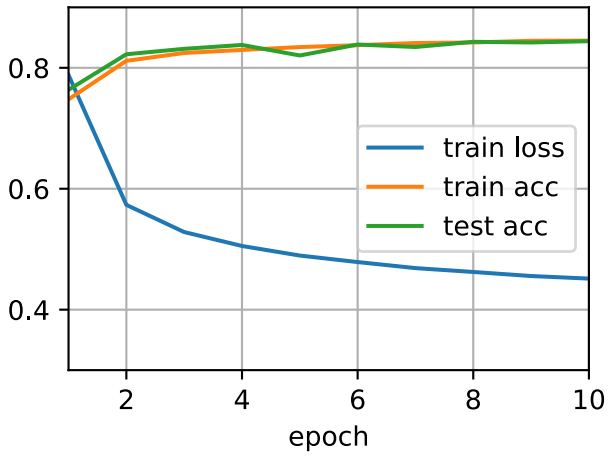
```
        display.clear_output(wait=True)
```

The training function then runs multiple epochs and visualize the training progress.

```
# Saved in the d2l package for later use
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):
    trains, test_accs = [], []
    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,))
```

Again, we use the minibatch stochastic gradient descent to optimize the loss function of the model. Note that the number of epochs (`num_epochs`), and learning rate (`lr`) are both adjustable hyper-parameters. By changing their values, we may be able to increase the classification accuracy of the model. In practice we will want to split our data three ways into training, validation, and test data, using the validation data to choose the best values of our hyperparameters.

```
num_epochs, lr = 10, 0.1
updater = lambda batch_size: d2l.sgd([W, b], lr, batch_size)
train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)
```



5.6.7 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 model predictions (second line of text output).

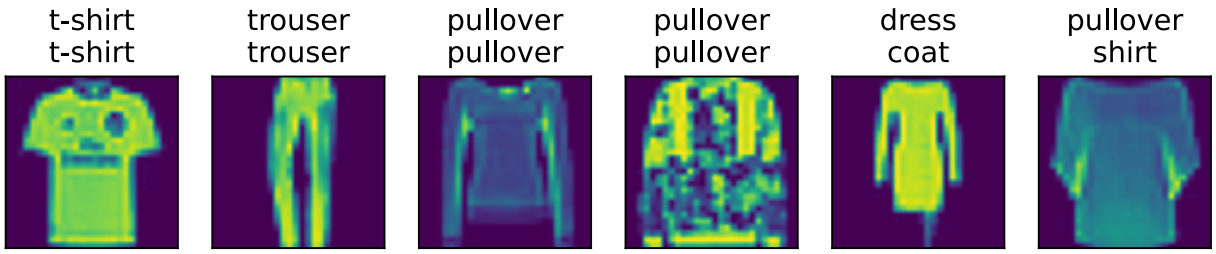
```
# Saved in the d2l package for later use
def predict_ch3(net, test_iter, n=6):
    for X, y in test_iter:
        break
    trues = d2l.get_fashion_mnist_labels(y)
    preds = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1))
```

(continues on next page)

(continued from previous page)

```
    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)
```



5.6.8 Summary

With softmax regression, we can train models for multi-category classification. The training loop 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.

5.6.9 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 is 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. E.g. 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?

5.6.10 Scan the QR Code to Discuss[61]

5.7 Concise Implementation of Softmax Regression

Just as Gluon made it much easier to implement linear regression in Section 5.3, we will find it similarly (or possibly more) convenient for implementing classification models. Again, we begin with our import ritual.

[61] https://discuss.mxnet.io/t/2336

```
import d2l
from mxnet import gluon, init, npx
from mxnet.gluon import nn
npx.set_np()
```

Let us stick with the Fashion-MNIST dataset and keep the batch size at 256 as in the last section.

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

5.7.1 Initialize Model Parameters

As mentioned in Section 5.4, the output layer of softmax regression is a fully-connected (`Dense`) layer. Therefore, to implement our model, we just need to add one `Dense` 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))
```

5.7.2 The Softmax

In the previous example, 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 (as discussed in Section 18.9). Recall that the softmax function calculates $\hat{y}_j = \frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}$, where $\hat{y}_j$ is the j^{th} element of `yhat` and z_j is the j^{th} element of the input `y_linear` variable, as computed by the softmax.

If some of the z_i are very large (i.e., very positive), then e^{z_i} might be larger than the largest number we can have for certain types of `float` (i.e., overflow). This would make the denominator (and/or numerator) `inf` and we wind up encountering either 0, `inf`, or `nan` 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 $\max(z_i)$ from all z_i before proceeding with the `softmax` calculation. You can verify that this shifting of each z_i by constant factor does not change the return value of `softmax`.

After the subtraction and normalization step, it might be that possible that some z_j have large negative values and thus that the corresponding e^{z_j} 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 not-a-number (`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 avoided calculating e^{z_j} and can instead z_j directly due to the canceling

in $\log(\exp(\cdot))$.

$$\begin{aligned}\log\left(\hat{y}_j\right) &= \log\left(\frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}\right) \\ &= \log\left(e^{z_j}\right) - \log\left(\sum_{i=1}^{n} e^{z_i}\right) \\ &= z_j - \log\left(\sum_{i=1}^{n} e^{z_i}\right)\end{aligned} \tag{5.7.1}$$

We will want to keep the conventional softmax function handy in case we ever want to evaluate the probabilities output 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 softmax_cross_entropy loss function, which does smart things like the log-sum-exp trick (see on Wikipedia[62]).

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

5.7.3 Optimization Algorithm

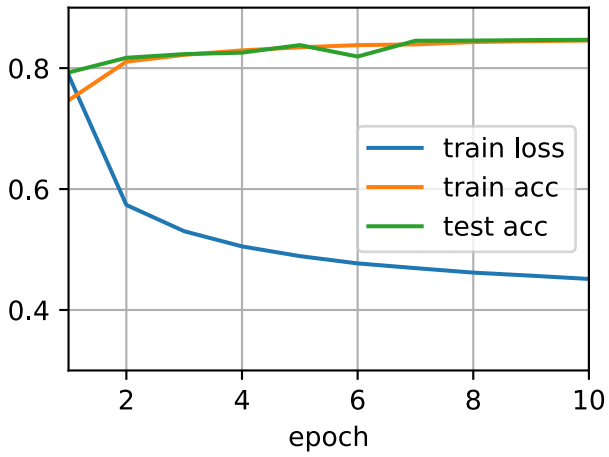
Here, we use minibatch stochastic gradient descent with a learning rate of 0.1 as the optimization algorithm. Note that this is the same as we applied in the linear regression example and it illustrates the general applicability of the optimizers.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

5.7.4 Training

Next we call the training function defined in the last section to train a model.

```
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



As before, this algorithm converges to a solution that achieves an accuracy of 83.7%, albeit this time with fewer lines of code than before. Note that in many cases, Gluon takes additional precautions beyond these most well-known tricks to

[62] https://en.wikipedia.org/wiki/LogSumExp

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.

5.7.5 Exercises

1. Try adjusting the hyper-parameters, such as batch size, epoch, and learning rate, to see what the results are.
2. Why might the test accuracy decrease again after a while? How could we fix this?

5.7.6 Scan the QR Code to Discuss[63]

[63] https://discuss.mxnet.io/t/2337

CHAPTER

SIX

MULTILAYER PERCEPTRONS

In this chapter, we will introduce your first truly *deep* networks. The simplest deep networks are called multilayer perceptrons, and they consist of many 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 capacity control. To help you combat these problems, we will introduce regularization techniques such as dropout and weight decay. We will also discuss issues relating to numerical stability and parameter initialization that are key to successfully training deep networks. Throughout, we focus on applying models to real data, aiming to give the reader a firm grasp not just of the concepts but also of the practice of using deep networks. We punt matters relating to the computational performance, scalability and efficiency of our models to subsequent chapters.

6.1 Multilayer Perceptron

In the previous chapters, we showed how you could implement multiclass logistic regression (also called softmax regression) for classifying images of clothing into the 10 possible categories. To get there, we had to learn how to wrangle data, coerce our outputs into a valid probability distribution (via `softmax`), how to apply an appropriate loss function, and how to optimize over our parameters. Now that we've covered these preliminaries, we are free to focus our attention on the more exciting enterprise of designing powerful models using deep neural networks.

6.1.1 Hidden Layers

Recall that for linear regression and softmax regression, we mapped our inputs directly to our outputs via a single linear transformation:

$$\hat{\mathbf{o}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{6.1.1}$$

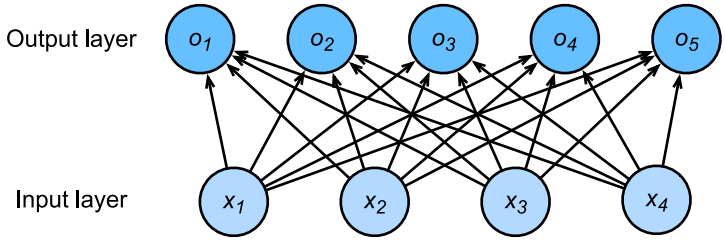


Fig. 6.1.1: Single layer perceptron with 5 output units.

If our labels really were related to our input data by an approximately linear function, then this approach would be perfect. But linearity is a *strong assumption*. Linearity implies that for whatever target value we are trying to predict, increasing

the value of each of our inputs should either drive the value of the output up or drive it down, irrespective of the value of the other inputs.

Sometimes this makes sense! Say we are trying to predict whether an individual will or will not repay a loan. We might reasonably imagine that all else being equal, an applicant with a higher income would be more likely to repay than one with a lower income. In these cases, linear models might perform well, and they might even be hard to beat.

But what about classifying images in FashionMNIST? Should increasing the intensity of the pixel at location (13,17) always increase the likelihood that the image depicts a pocketbook? That seems ridiculous because we all know that you cannot make sense out of an image without accounting for the interactions among pixels.

From one to many

As another case, consider trying to classify images based on whether they depict *cats* or *dogs* given black-and-white images.

If we use a linear model, we'd basically be saying that for each pixel, increasing its value (making it more white) must always increase the probability that the image depicts a dog or must always increase the probability that the image depicts a cat. We would be making the absurd assumption that the only requirement for differentiating cats vs. dogs is to assess how bright they are. That approach is doomed to fail in a work that contains both black dogs and black cats, and both white dogs and white cats.

Teasing out what is depicted in an image generally requires allowing more complex relationships between our inputs and outputs. Thus we need models capable of discovering patterns that might be characterized by interactions among the many features. We can over come 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 layers of neurons on top of each other. Each layer feeds into the layer above it, until we generate an output. This architecture is commonly called a *multilayer perceptron*, often abbreviated as *MLP*. The neural network diagram for an MLP looks like this:

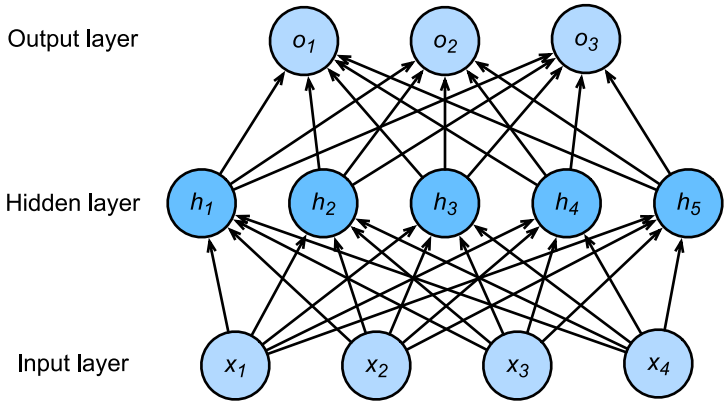


Fig. 6.1.2: Multilayer perceptron with hidden layers. This example contains a hidden layer with 5 hidden units in it.

The multilayer perceptron above has 4 inputs and 3 outputs, and the hidden layer in the middle contains 5 hidden units. Since the input layer does not involve any calculations, building this network would consist of implementing 2 layers of computation. The neurons in the input layer are fully connected to the inputs in the hidden layer. Likewise, the neurons in the hidden layer are fully connected to the neurons in the output layer.

From linear to nonlinear

We can write out the calculations that define this one-hidden-layer MLP in mathematical notation as follows:

$$\begin{aligned}\mathbf{h} &= \mathbf{W}_1\mathbf{x} + \mathbf{b}_1\\ \mathbf{o} &= \mathbf{W}_2\mathbf{h} + \mathbf{b}_2\\ \hat{\mathbf{y}} &= \mathrm{softmax}(\mathbf{o})\end{aligned} \tag{6.1.2}$$

By adding another layer, we have added two new sets of parameters, but what have we gained in exchange? In the model defined above, we do not achieve anything for our troubles!

That is because our hidden units are just a linear function of the inputs and the outputs (pre-softmax) are just a linear function of the hidden units. A linear function of a linear function is itself a linear function. That means that for any values of the weights, we could just collapse out the hidden layer yielding an equivalent single-layer model using $\mathbf{W} = \mathbf{W}_2\mathbf{W}_1$ and $\mathbf{b} = \mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2$.

$$\mathbf{o} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2 = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2) = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{6.1.3}$$

In order to get a benefit from multilayer architectures, we need another key ingredient—a nonlinearity σ to be applied to each of the hidden units after each layer's linear transformation. The most popular choice for the nonlinearity these days is the rectified linear unit (ReLU) $\max(x, 0)$. After incorporating these non-linearities it becomes impossible to merge layers.

$$\begin{aligned}\mathbf{h} &= \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\\ \mathbf{o} &= \mathbf{W}_2\mathbf{h} + \mathbf{b}_2\\ \hat{\mathbf{y}} &= \mathrm{softmax}(\mathbf{o})\end{aligned} \tag{6.1.4}$$

Clearly, we could continue stacking such hidden layers, e.g., $\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$ and $\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$ on top of each other to obtain a true multilayer perceptron.

Multilayer perceptrons can account for complex interactions in the inputs because the hidden neurons depend on the values of each of the inputs. It's easy to design a hidden node that does arbitrary computation, such as, for instance, logical operations on its inputs. Moreover, for certain choices of the activation function it's widely known that multilayer perceptrons are universal approximators. That means that even for a single-hidden-layer neural network, with enough nodes, and the right set of weights, we can model any function at all! *Actually learning that function is the hard part.*

Moreover, just because a single-layer network *can* learn any function does not mean that you should try to solve all of your problems with single-layer networks. It turns out that we can approximate many functions much more compactly if we use deeper (vs wider) neural networks. We'll get more into the math in a subsequent chapter, but for now let's actually build an MLP. In this example, we'll implement a multilayer perceptron with two hidden layers and one output layer.

Vectorization and minibatch

As before, by the matrix $\mathbf{X}$, we denote a minibatch of inputs. The calculations to produce outputs from an MLP with two hidden layers can thus be expressed:

$$\begin{aligned}\mathbf{H}_1 &= \sigma(\mathbf{W}_1\mathbf{X} + \mathbf{b}_1)\\ \mathbf{H}_2 &= \sigma(\mathbf{W}_2\mathbf{H}_1 + \mathbf{b}_2)\\ \mathbf{O} &= \mathrm{softmax}(\mathbf{W}_3\mathbf{H}_2 + \mathbf{b}_3)\end{aligned} \tag{6.1.5}$$

With some abuse of notation, we define the nonlinearity σ to apply to its inputs on a row-wise fashion, i.e., one observation at a time. Note that we are also using the notation for *softmax* in the same way to denote a row-wise operation. Often, as in this chapter, the activation functions that we apply to hidden layers are not merely row-wise, but component wise. That means that after computing the linear portion of the layer, we can calculate each nodes activation without looking at the values taken by the other hidden units. This is true for most activation functions (the batch normalization operation will be introduced in Section 9.5 is a notable exception to that rule).

```
%matplotlib inline
import d2l
from mxnet import autograd, np, npx
npx.set_np()
```

6.1.2 Activation Functions

Because they are so fundamental to deep learning, before going further, let us take a brief look at some common activation functions.

ReLU Function

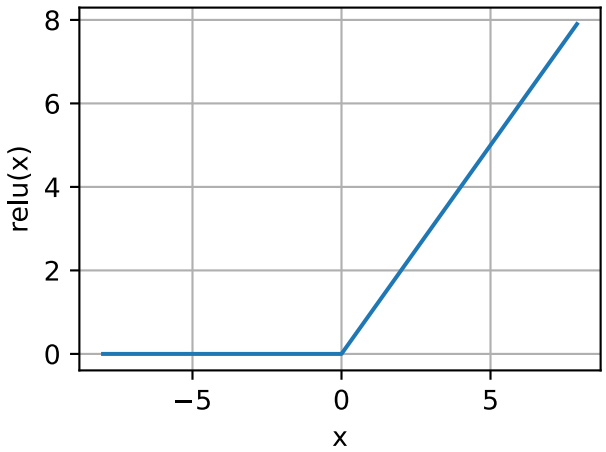
As stated above, the most popular choice, due to its simplicity of implementation and its efficacy in training is the rectified linear unit (ReLU). ReLUs provide a very simple nonlinear transformation. Given the element z, the function is defined as the maximum of that element and 0.

$$\text{ReLU}(z) = \max(z, 0). \tag{6.1.6}$$

It can be understood that the ReLU function retains only positive elements and discards negative elements (setting those nodes to 0). To get a better idea of what it looks like, we can plot it.

Because it is used so commonly, NDarray supports the `relu` function as a basic native operator. As you can see, the activation function is piece-wise linear.

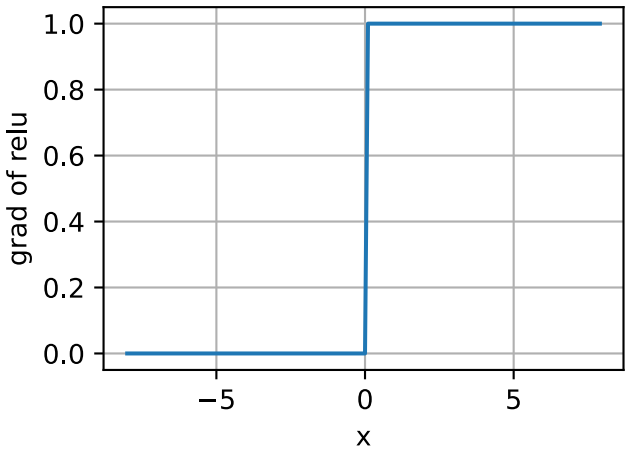
```
x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.relu(x)
d2l.set_figsize((4, 2.5))
d2l.plot(x, y, 'x', 'relu(x)')
```



When the input is negative, the derivative of ReLU function is 0 and when the input is positive, the derivative of 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 go with the left-hand-side (LHS) 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. See the derivative of the ReLU function plotted below.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of relu')
```



Note that there are many variants to the ReLU function, such as the parameterized ReLU (pReLU) of He et al., 2015[64]. This variation adds a linear term to the ReLU, so some information still gets through, even when the argument is negative.

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x) \tag{6.1.7}$$

The reason for using the 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 reduces the issue of the vanishing gradient problem (more on this later).

Sigmoid Function

The sigmoid function transforms its inputs which take values in $\mathbb{R}$ to 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).

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{6.1.8}$$

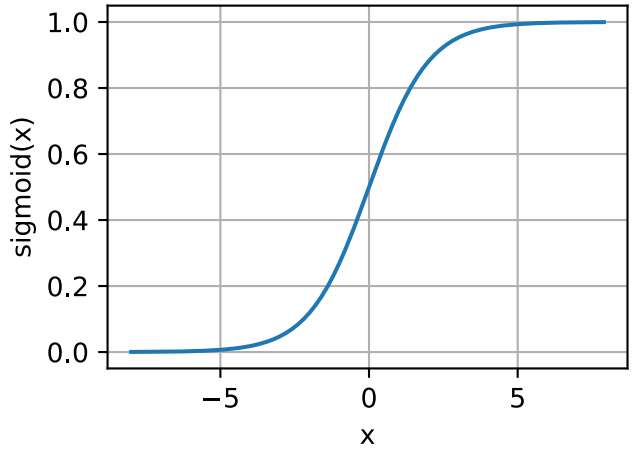
In the earliest neural networks, scientists were interested in modeling biological neurons which either *fire* or *do not fire*. Thus the pioneers of this field, going all the way back to McCulloch and Pitts in the 1940s, were focused on thresholding units. A thresholding function takes either value 0 (if the input is below the threshold) or value 1 (if 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 common 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) but the sigmoid has mostly been replaced by the simpler and easier to train ReLU for most use in hidden layers. In the "Recurrent Neural Network" chapter, we will describe how sigmoid units can be used to control the flow of information in a neural network thanks to its capacity to transform the value range between 0 and 1.

See the sigmoid function plotted below. When the input is close to 0, the sigmoid function approaches a linear transformation.

[64] https://arxiv.org/abs/1502.01852

```
with autograd.record():
    y = npx.sigmoid(x)
d2l.plot(x, y, 'x', 'sigmoid(x)')
```

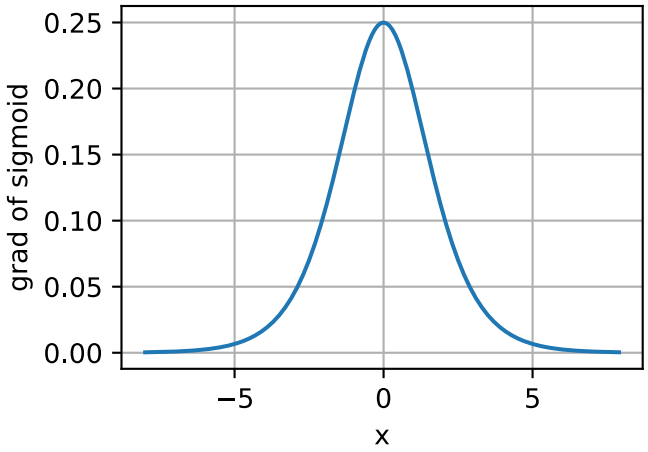


The derivative of sigmoid function is given by the following equation:

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x)\,(1-\text{sigmoid}(x))\,. \tag{6.1.9}$$

The derivative of 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')
```



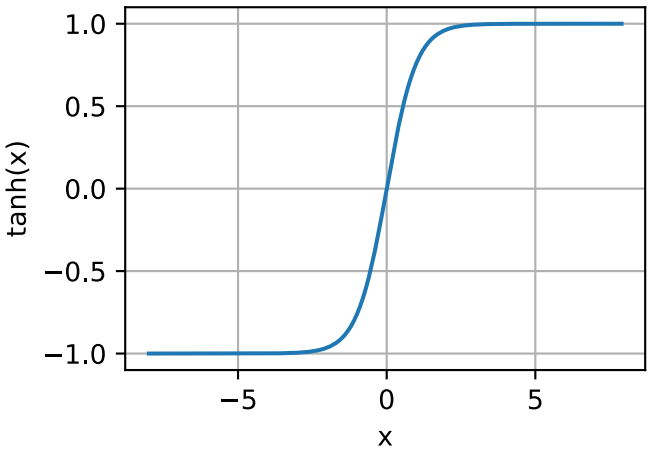
Tanh Function

Like the sigmoid function, the tanh (Hyperbolic Tangent) function also squashes its inputs, transforms them into elements on the interval between -1 and 1:

$$\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}. \tag{6.1.10}$$

We plot the tanh function blow. Note that as the input nears 0, the tanh function approaches a linear transformation. Although the shape of the function is similar to 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)')
```

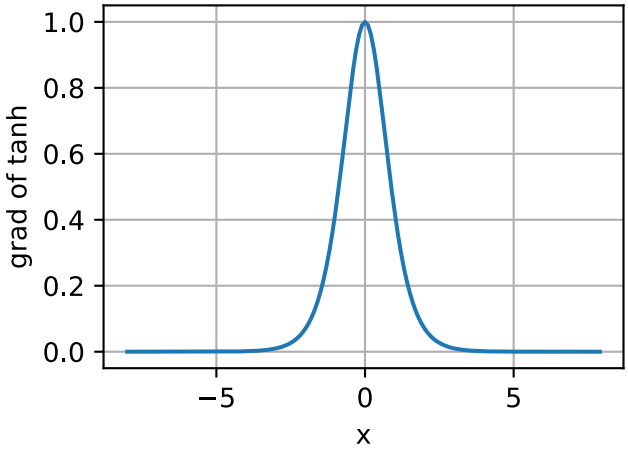


The derivative of the Tanh function is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x). \tag{6.1.11}$$

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')
```



In summary, we now know how to incorporate nonlinearities to build expressive multilayer neural network architectures. As a side note, your knowledge now already puts you in command of the state of the art in deep learning, circa 1990.

In fact, you have an advantage over anyone working the 1990s, because you can leverage powerful open-source deep learning frameworks to build models rapidly, using only a few lines of code. Previously, getting these nets training required researchers to code up thousands of lines of C and Fortran.

6.1.3 Summary

- The multilayer perceptron 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.

6.1.4 Exercises

1. Compute the derivative of the tanh and the pReLU activation function.
2. Show that a multilayer perceptron using only ReLU (or pReLU) constructs a continuous piecewise linear function.
3. Show that $\tanh(x) + 1 = 2\text{sigmoid}(2x)$.
4. Assume we have a multilayer perceptron *without* nonlinearities between the layers. In particular, assume that we have d input dimensions, d output dimensions and that one of the layers had only $d/2$ dimensions. Show that this network is less expressive (powerful) than a single layer perceptron.
5. Assume that we have a nonlinearity that applies to one minibatch at a time. What kinds of problems do you expect this to cause?

6.1.5 Scan the QR Code to Discuss[65]

6.2 Implementation of Multilayer Perceptron from Scratch

Now that we know how multilayer perceptrons (MLPs) work in theory, let us implement them. First, we import the required packages.

```
import d2l
from mxnet import np, npx, gluon
npx.set_np()
```

To compare against the results we previously achieved with vanilla softmax regression, we continue to use the Fashion-MNIST image classification dataset.

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

[65] https://discuss.mxnet.io/t/2338

6.2.1 Initialize Model Parameters

Recall that this dataset contains 10 classes and that each image consists of a $28 \times 28 = 784$ grid of pixel values. Since we will be discarding the spatial structure (for now), we can just think of this as a classification dataset with 784 input features and 10 classes. In particular we will implement our MLP with one hidden layer and 256 hidden units. Note that we can regard both of these choices as *hyperparameters* that could be set based on performance on validation data. Typically, we will choose layer widths as powers of 2 to make everything align nicely in memory.

Again, we will represent our parameters with several `ndarrays`. Note that we now have one weight matrix and one bias vector *per layer*. As always, we must call `attach_grad` to allocate memory for the gradients 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()
```

6.2.2 Activation Function

To make sure we know how everything works, we will use the `maximum` function to implement ReLU ourselves, instead of invoking `npx.relu` directly.

```
def relu(X):
    return np.maximum(X, 0)
```

6.2.3 The model

As in softmax regression, we will `reshape` each 2D image into a flat vector of length `num_inputs`. Finally, we can 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
```

6.2.4 The Loss Function

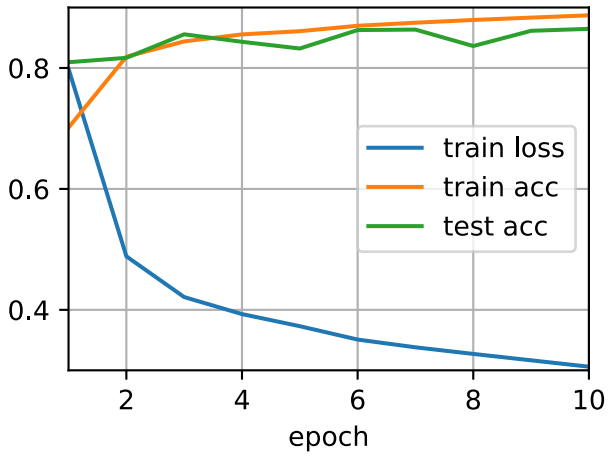
For better numerical stability and because we already know how to implement softmax regression completely from scratch in Section 5.6, we will use Gluon's integrated function for calculating the softmax and cross-entropy loss. Recall that we discussed some of these intricacies in Section 6.1. We encourage the interested reader to examing the source code for `mxnet.gluon.loss.SoftmaxCrossEntropyLoss` for more details.

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

6.2.5 Training

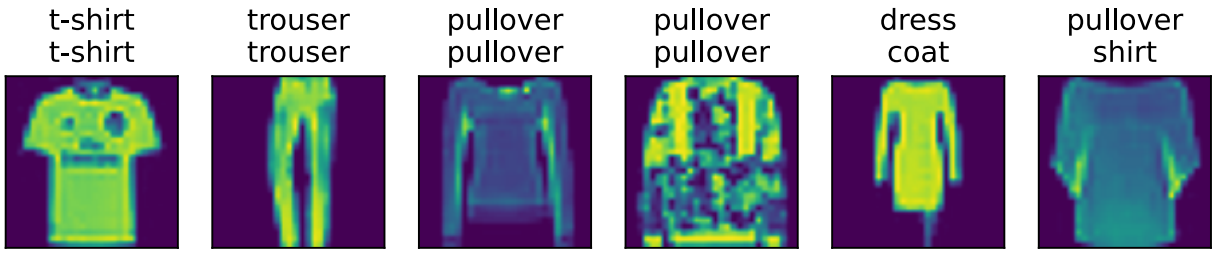
Steps for training the MLP are no different than for softmax regression. In the `d2l` package, we directly call the `train_ch3` function, whose implementation was introduced in Section 5.6. We set the number of epochs to 10 and the learning rate to 0.5.

```
num_epochs, lr = 10, 0.5
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
              lambda batch_size: d2l.sgd(params, lr, batch_size))
```



To see how well we did, let us apply the model to some test data. If you are interested, compare the result to corresponding linear model in Section 5.6.

```
d2l.predict_ch3(net, test_iter)
```



This looks a bit better than our previous result, a good sign that we are on the right path.

6.2.6 Summary

We saw that implementing a simple MLP is easy, even when done manually. That said, with a large number of layers, this can get messy (e.g., naming and keeping track of the model parameters, etc).

6.2.7 Exercises

1. Change the value of the hyperparameter `num_hiddens` in order to see how this hyperparameter influences your results.

2. Try adding a new hidden layer to see how it affects the results.
3. How does changing the learning rate change the result?
4. What is the best result you can get by optimizing over all the parameters (learning rate, iterations, number of hidden layers, number of hidden units per layer)?

6.2.8 Scan the QR Code to Discuss[66]

6.3 Concise Implementation of Multilayer Perceptron

Now that we learned how multilayer perceptrons (MLPs) work in theory, let us implement them. We begin, as always, by importing modules.

```
import d2l
from mxnet import gluon, npx, init
from mxnet.gluon import nn
npx.set_np()
```

6.3.1 The Model

The only difference from our softmax regression implementation is that we add two `Dense` (fully-connected) layers instead of one. The first is our hidden layer, which has *256* hidden units and uses the ReLU activation function.

```
net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'),
        nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

Again, note that as always, Gluon automatically infers the missing input dimensions to each layer.

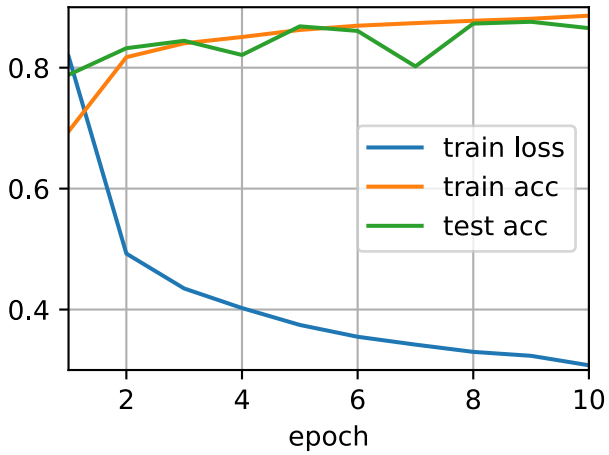
Training the model follows the exact same steps as in our softmax regression implementation.

```
batch_size, num_epochs = 256, 10
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
loss = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

6.3.2 Exercises

1. Try adding a few more hidden layers to see how the result changes.

[66] https://discuss.mxnet.io/t/2339



2. Try out different activation functions. Which ones work best?
3. Try out different initializations of the weights.

6.3.3 Scan the QR Code to Discuss[67]

6.4 Model Selection, Underfitting and Overfitting

As machine learning scientists, our goal is to discover general patterns. Say, for example, that we wish to learn the pattern that associates genetic markers with the development of dementia in adulthood. It is easy enough to memorize our training set. Each person's genes uniquely identify them, not just among people represented in our dataset, but among all people on earth!

Given the genetic markers representing some person, we do not want our model to simply recognize "oh, that is Bob", and then output the classification, say among {*dementia*, *mild cognitive impairment*, *healthy*}, that corresponds to Bob. Rather, our goal is to discover patterns that capture regularities in the underlying population from which our training set was drawn. If we are successfully 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. And more often we have to learn from thousands or tens of thousands. In a large hospital system we might access hundreds of thousands of medical records. With finite samples, we always run the risk that we might discover *apparent* associations that turn out not to hold up when we collect more data.

Let's consider an extreme pathological case. Imagine that you want to learn to predict which people will repay their loans. A lender hires you as a data scientist to investigate, handing over the complete files on 100 applicants, 5 of which defaulted on their loans within 3 years. Realistically, the files might include hundreds of potential features, including

[67] https://discuss.mxnet.io/t/2340

income, occupation, credit score, length of employment etc. Moreover, say that they additionally hand over video footage of each applicant's interview with their lending agent.

Now suppose that after featurizing the data into an enormous design matrix, you discover that of the 5 applicants who default, all of them were wearing blue shirts during their interviews, while only 40% of general population wore blue shirts. There is a good chance that if you train a predictive model to predict default, it might rely upon blue-shirt-wearing as an important feature.

Even if in fact defaulters were no more likely to wear blue shirts than people in the general population, there's a $.4^5 = .01$ probability that we would observe all five defaulters wearing blue shirts. With just 5 positive examples of defaults and hundreds or thousands of features, we would probably find a large number of features that appear to be perfectly predictive of our labor just due to random chance. With an unlimited amount of data, we would expect these *spurious* associations to eventually disappear. But we seldom have that luxury.

The phenomena 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 hyper-parameters during the experiment, you might have noticed that with enough nodes, layers, and training epochs, the model can eventually reach perfect accuracy on the training set, even as the accuracy on test data deteriorates.

6.4.1 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 points drawn from the same underlying data distribution as our original sample.

Problematically, *we can never calculate the generalization error exactly*. That is because the imaginary 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 points 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 her abilities using exams from previous years. Nonetheless, doing well on past exams is no guarantee that she 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 gray scale values, then there are 256^{784} possible images. That means that there are far more low-res grayscale thumbnail-sized images than there are atoms in the universe. Even if we could encounter this data, we could never afford to store the lookup table.

Lastly, 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. No matter what algorithm we come up with, because 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 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.

Statistical Learning Theory

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[68], 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[69] extended this theory to more general classes of functions. This work laid the foundations of Statistical Learning Theory[70].

In the *standard supervised learning setting*, which we have addressed up until now and will stick throughout most of this book, we assume that both the training data and the test data are drawn *independently* from *identical* distributions (commonly called the i.i.d. assumption). This means that the process that samples our data has no *memory*. The 2nd example drawn and the 3rd drawn are no more correlated than the 2nd and the 2-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, 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 useful tools for face recognition, speech recognition, language translation, etc.

Other violations are sure to cause trouble. Imagine, for example, if we tried 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 and volumes, 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 vexes the greatest minds in learning theory.

When we train our models, we attempt searching 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 the 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.

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 steps as more complex, and one subject to *early stopping* as less complex.

It can be difficult to compare the complexity among members of substantially different model classes (say a decision tree versus a neural network). 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[71] of a scientific theory: a theory is good if it fits data and if there are specific tests which can be used to disprove it. This is

[68] https://en.wikipedia.org/wiki/Glivenko%E2%80%93Cantelli_theorem
[69] https://en.wikipedia.org/wiki/Vapnik%E2%80%93Chervonenkis_theory
[70] https://en.wikipedia.org/wiki/Statistical_learning_theory
[71] https://en.wikipedia.org/wiki/Falsifiability

important since all statistical estimation is post hoc[72], 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 chapter, to give you some intuition, we'll 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 over fitting.
3. The number of training examples. It's 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.

6.4.2 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 multilayer perceptrons 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 set.

Validation Data Set

In principle we should not touch our test set until after we have chosen all our hyper-parameters. 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.

The common practice to address this problem is to split our data three ways, incorporating a *validation set* in addition to the training and test sets.

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 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 is really the validation accuracy and not a true test set accuracy. The good news is that we do not need too much data in the validation set. The uncertainty in our estimates can be shown to be of the order of $O(n^{-\frac{1}{2}})$.

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 $K-1$

[72] https://en.wikipedia.org/wiki/Post_hoc

subsets and validating on a different subset (the one not used for training in that round). Finally, the training and validation error rates are estimated by averaging over the results from the K experiments.

6.4.3 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 given 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

$$\hat{y} = \sum_{i=0}^{d} x^i w_i \tag{6.4.1}$$

to estimate the labels y. This is just a linear regression problem where our features are given by the powers of x, the w_i given the model's weights, 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 points each have a distinct value of x, a polynomial function with degree equal to the number of data points can fit the training set perfectly. We visualize the relationship between polynomial degree and under- vs over-fitting below.

Data Set 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 hurts. 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 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.

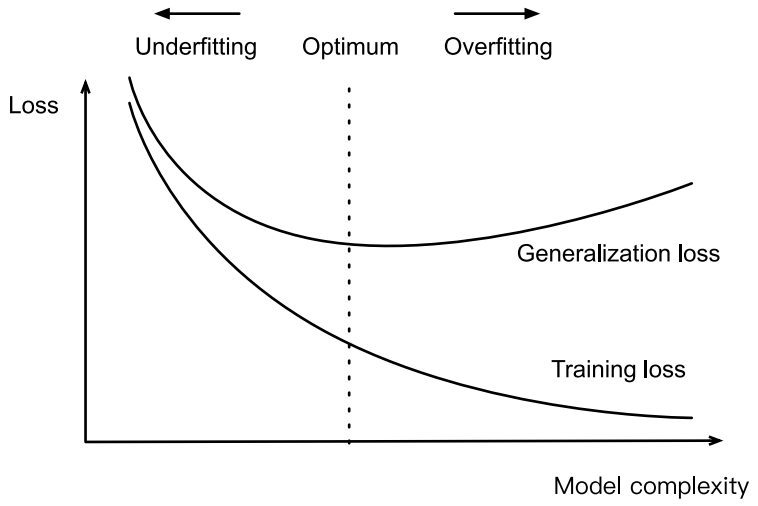


Fig. 6.4.1: Influence of Model Complexity on Underfitting and Overfitting

6.4.4 Polynomial Regression

We can now explore these concepts interactively by fitting polynomials to data. To get started we will import our usual packages.

```
import d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
npx.set_np()
```

Generating Data Sets

First we need data. Given x, we will use the following cubic polynomial to generate the labels on training and test data:

$$y = 5 + 1.2x - 3.4\frac{x^2}{2!} + 5.6\frac{x^3}{3!} + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1) \tag{6.4.2}$$

The noise term ϵ obeys a normal distribution with a mean of 0 and a standard deviation of 0.1. We will synthesize 100 samples each for the training set and test set.

```
maxdegree = 20  # Maximum degree of the polynomial
n_train, n_test = 100, 100  # Training and test dataset sizes
true_w = np.zeros(maxdegree)  # 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))
features = np.random.shuffle(features)
poly_features = np.power(features, np.arange(maxdegree).reshape(1, -1))
poly_features = poly_features / (
    npx.gamma(np.arange(maxdegree) + 1).reshape(1, -1))
labels = np.dot(poly_features, true_w)
labels += np.random.normal(scale=0.1, size=labels.shape)
```

For optimization, we typically want to avoid very large values of gradients, losses, etc. This is why the monomials stored in `poly_features` are rescaled from x^i to $\frac{1}{i!}x^i$. It allows us to avoid very large values for large exponents i. Factorials

are implemented in Gluon using the Gamma function, where $n! = \Gamma(n+1)$.

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([[1.5094751],
        [1.9676613]]),
 array([[1.0000000e+00, 1.5094751e+00, 1.1392574e+00, 5.7322693e-01,
         2.1631797e-01, 6.5305315e-02, 1.6429458e-02, 3.5428370e-03,
         6.6847802e-04, 1.1211676e-04, 1.6923748e-05, 2.3223611e-06,
         2.9212887e-07, 3.3920095e-08, 3.6572534e-09, 3.6803552e-10,
         3.4721288e-11, 3.0829944e-12, 2.5853909e-13, 2.0539909e-14],
        [1.0000000e+00, 1.9676613e+00, 1.9358451e+00, 1.2696959e+00,
         6.2458295e-01, 2.4579351e-01, 8.0606394e-02, 2.2658013e-02,
         5.5729118e-03, 1.2184002e-03, 2.3973989e-04, 4.2884261e-05,
         7.0318083e-06, 1.0643244e-06, 1.4958786e-07, 1.9622549e-08,
         2.4131586e-09, 2.7931044e-10, 3.0532687e-11, 3.1619987e-12]]),
 array([6.1804767, 7.7595935]))
```

Defining, Training and Testing Model

Let first implement a function to evaluate the loss on a given data.

```
# Saved in the d2l package for later use
def evaluate_loss(net, data_iter, loss):
    """Evaluate the loss of a model on the given dataset"""
    metric = d2l.Accumulator(2)  # sum_loss, num_examples
    for X, y in data_iter:
        metric.add(loss(net(X), y).sum(), y.size)
    return metric[0] / metric[1]
```

Now define the training function.

```
def train(train_features, test_features, train_labels, test_labels,
          num_epochs=1000):
    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(1, num_epochs+1):
        d2l.train_epoch_ch3(net, train_iter, loss, trainer)
        if epoch % 50 == 0:
```

(continues on next page)

(continued from previous page)

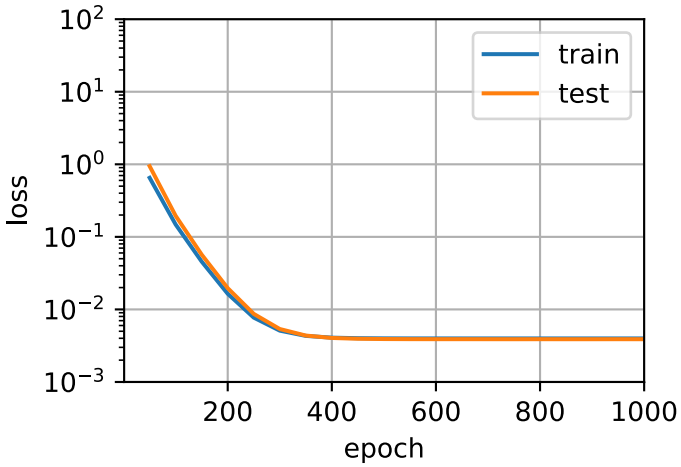
```
            animator.add(epoch, (evaluate_loss(net, train_iter, loss),
                                 evaluate_loss(net, test_iter, loss)))
    print('weight:', net[0].weight.data().asnumpy())
```

Third-order Polynomial Function Fitting (Normal)

We will begin by first using a third-order polynomial function with the same order as the data generation function. The results show that this model's training error rate when using the testing dataset is low. The trained 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, x^3 from the polynomial
# features
train(poly_features[:n_train, 0:4], poly_features[n_train:, 0:4],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 5.0161023  1.1760497 -3.4186537  5.6422276]]
```

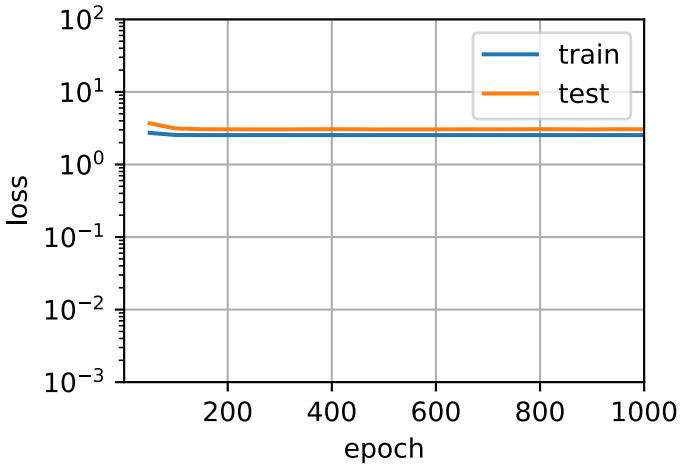


Linear Function Fitting (Underfitting)

Let's take another look at linear function fitting. After the decline in the early epoch, it becomes difficult to further decrease this model's training error rate. After the last epoch iteration has been completed, the training error rate is still high. When used to fit non-linear patterns (like the third-order polynomial function here) linear models are liable to underfit.

```
# Pick the first four dimensions, i.e., 1, x from the polynomial features
train(poly_features[:n_train, 0:3], poly_features[n_train:, 0:3],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 4.894285   4.109313  -2.3608558]]
```



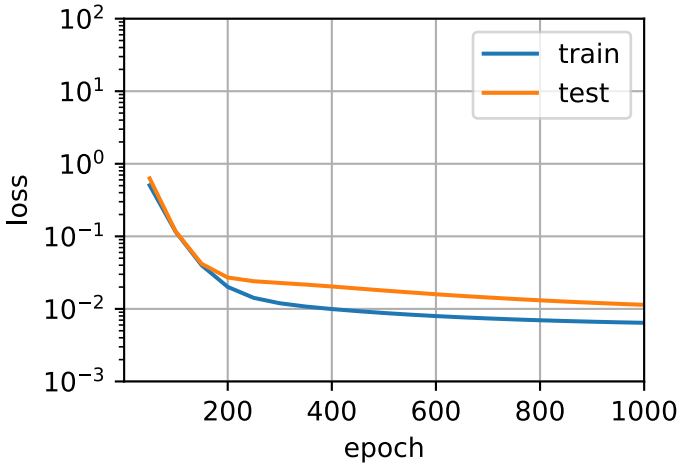
Insufficient Training (Overfitting)

Now let us try to train the model using a polynomial of too high degree. Here, there is insufficient data to learn that the higher-degree coefficients should have values close to zero. As a result, our overly-complex model is far too susceptible to being influenced by noise in the training data. Of course, our training error will now be low (even lower than if we had the right model!) but our test error will be high.

Try out different model complexities (`n_degree`) and training set sizes (`n_subset`) to gain some intuition of what is happening.

```
n_subset = 100  # Subset of data to train on
n_degree = 20   # Degree of polynomials
train(poly_features[1:n_subset, 0:n_degree],
      poly_features[n_train:, 0:n_degree], labels[1:n_subset],
      labels[n_train:])
```

```
weight: [[ 4.983486    1.3045875  -3.2569666   5.0300164  -0.4048594   1.6138743
   0.11082392  0.3047985   0.03454423  0.04253425 -0.03632538  0.05625797
   0.06353628  0.02527205 -0.00739609 -0.00711016  0.04849716  0.06699993
   0.0279271  -0.05373173]]
```



In later chapters, we will continue to discuss overfitting problems and methods for dealing with them, such as weight decay and dropout.

6.4.5 Summary

- Since the generalization error rate cannot be estimated based on the training error rate, simply minimizing the training error rate will not necessarily mean a reduction in the generalization error rate. Machine learning models need to be careful to safeguard against overfitting such 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 the model is not able to reduce the training error rate while overfitting is a result of the model training error rate being much lower than the testing dataset rate.
- We should choose an appropriately complex model and avoid using insufficient training samples.

6.4.6 Exercises

1. Can you solve the polynomial regression problem exactly? Hint - use linear algebra.
2. Model selection for polynomials
    - Plot the training error vs. model complexity (degree of the polynomial). What do you observe?
    - Plot the test error in this case.
    - Generate the same graph as a function of the amount of data?
3. What happens if you drop the normalization of the polynomial features x^i by $1/i!$. Can you fix this in some other way?
4. What degree of polynomial do you need to reduce the training error to 0?
5. Can you ever expect to see 0 generalization error?

6.4.7 Scan the QR Code to Discuss[73]

6.5 Weight Decay

Now that we have characterized the problem of overfitting and motivated the need for capacity control, we can begin discussing some of the popular techniques used to these ends in practice. Recall that we can always mitigate overfitting by going out and collecting more training data, that can be costly and time consuming, typically making it impossible in the short run. For now, let us assume that we have already obtained as much high-quality data as our resources permit and focus on techniques aimed at limiting the capacity of the function classes under consideration.

[73] https://discuss.mxnet.io/t/2341

In our toy example, we saw that we could control the complexity of a polynomial by adjusting its degree. However, most of machine learning does not consist of polynomial curve fitting. And moreover, even when we focus on polynomial regression, when we deal with high-dimensional data, manipulating model capacity by tweaking the degree d is problematic. To see why, note that for multivariate data we must generalize the concept of polynomials to include *monomials*, which are simply products of powers of variables. For example, $x_1^2 x_2$, and $x_3 x_5^2$ are both monomials of degree 3. The number of such terms with a given degree d blows up as a function of the degree d.

Concretely, for vectors of dimensionality D, the number of monomials of a given degree d is $\binom{D-1+d}{D-1}$. Hence, a small change in degree, even from say 1 to 2 or 2 to 3 would entail a massive blowup in the complexity of our model. Thus, tweaking the degree is too blunt a hammer. Instead, we need a more fine-grained tool for adjusting function complexity.

6.5.1 Squared Norm Regularization

Weight decay (commonly called *L2* regularization), might be the most widely-used technique for regularizing parametric machine learning models. The basic intuition behind weight decay is the notion that among all functions f, the function $f = 0$ is the simplest. Intuitively, we can then measure functions by their proximity to 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, e.g., in functional analysis and the theory of Banach spaces are devoted to answering this issue.

For our present purposes, a very simple interpretation will suffice: We will consider a linear function $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ to be simple if its weight vector is small. We can measure this via $||mathbf{w}||^2$. One way of keeping the weight vector small is to add its norm as a penalty term to the problem of minimizing the loss. Thus we replace our original objective, *minimize the prediction error on the training labels*, with new objective, *minimize the sum of the prediction error and the penalty term*. Now, if the weight vector becomes too large, our learning algorithm will find more profit in minimizing the norm $||\mathbf{w}||^2$ versus minimizing the training error. That is exactly what we want. To illustrate things in code, let us revive our previous example from Section 5.1 for linear regression. There, our loss was given by

$$l(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right)^2 . \tag{6.5.1}$$

Recall that $\mathbf{x}^{(i)}$ are the observations, $y^{(i)}$ are labels, and $(\mathbf{w}, b)$ are the weight and bias parameters respectively. To arrive at a new loss function that penalizes the size of the weight vector, we need to add $||mathbf{w}||^2$, but how much should we add? To address this, we need to add a new hyperparameter, that we will call the *regularization constant* and denote by λ:

$$l(\mathbf{w}, b) + \frac{\lambda}{2} \|\mathbf{w}\|^2 \tag{6.5.2}$$

This non-negatice parameter $\lambda \geq 0$ governs the amount of regularization. For $\lambda = 0$, we recover our original loss function, whereas for $\lambda > 0$ we ensure that $\mathbf{w}$ cannot grow too large. The astute reader might wonder why we are squaring the norm of the weight vector. We do this for two reasons. First, we do it for computational convenience. By squaring the L2 norm, we remove the square root, leaving the sum of squares of each component of the weight vector. This is convenient because it is easy to compute derivatives of a sum of terms (the sum of derivatives equals the derivative of the sum).

Moreover, you might ask, why the L2 norm in the first place and not the L1 norm, or some other distance function. In fact, several other choices are valid and are popular throughout statistics. While L2-regularized linear models constitute the classic *ridge regression* algorithm L1-regularizaed linear regression is a similarly fundamental model in statistics popularly known as *lasso regression*.

One mathematical reason for working with the L2 norm and not some other norm, is that it penalizes large components of the weight vector much more than it penalizes small ones. This encourages our learning algorithm to discover models which distribute their weight across a larger number of features, which might make them more robust in practice since they do not depend precariously on a single feature. The stochastic gradient descent updates for L2-regularied regression are as follows:

$$w \leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right) \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right), \tag{6.5.3}$$

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 0. That is why the method is sometimes called "weight decay": because the penalty term literally causes our optimization algorithm to *decay* the magnitude of the weight at each step of training. This is more convenient than having to pick the number of parameters as we did for polynomials. In particular, we now have a continuous mechanism for adjusting the complexity of f. Small values of λ correspond to unconstrained **w**, whereas large values of λ constrain **w** considerably. Since we do not want to have large bias terms either, we often add b^2 as a penalty, too.

6.5.2 High-dimensional Linear Regression

For high-dimensional regression it is difficult to pick the 'right' dimensions to omit. Weight-decay regularization is a much more convenient alternative. We will illustrate this below. First, we will generate some synthetic data as before

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01) \tag{6.5.4}$$

representing our label as a linear function of our inputs, corrupted by Gaussian noise with zero mean and variance 0.01. To observe the effects of overfitting more easily, we can make our problem high-dimensional, setting the data dimension to $d = 200$ and working with a relatively small number of training examples—here we will set the sample size to 20:

```
%matplotlib inline
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

n_train, n_test, num_inputs, batch_size = 20, 100, 200, 1
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)
```

6.5.3 Implementation from Scratch

Next, we will show how to implement weight decay from scratch. All we have to do here is to add the squared ℓ_2 penalty as an additional loss term added to the original target function. The squared norm penalty derives its name from the fact that we are adding the second power $\sum_i w_i^2$. The ℓ_2 is just one among an infinite class of norms call p-norms, many of which you might encounter in the future. In general, for some number p, the ℓ_p norm is defined as

$$\|\mathbf{w}\|_p^p := \sum_{i=1}^{d} |w_i|^p \tag{6.5.5}$$

Initialize Model Parameters

First, we will define a function to randomly initialize our model parameters and run `attach_grad` on each to allocate memory for the gradients we will calculate.

```
def init_params():
    w = np.random.normal(scale=1, size=(num_inputs, 1))
    b = np.zeros(1)
```

(continues on next page)

(continued from previous page)

```
    w.attach_grad()
    b.attach_grad()
    return [w, b]
```

Define ℓ_2 Norm Penalty

Perhaps the most convenient way to implement this penalty is to square all terms in place and summ them up. 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).

```
def l2_penalty(w):
    return (w**2).sum() / 2
```

Define Training and Testing

The following code defines how to train and test the model separately on the training dataset and the test dataset. Unlike the previous sections, here, the ℓ_2 norm penalty term is added when calculating the final loss function. The linear network and the squared loss have not changed since the previous chapter, so we will just import them via `d2l.linreg` and `d2l.squared_loss` to reduce clutter.

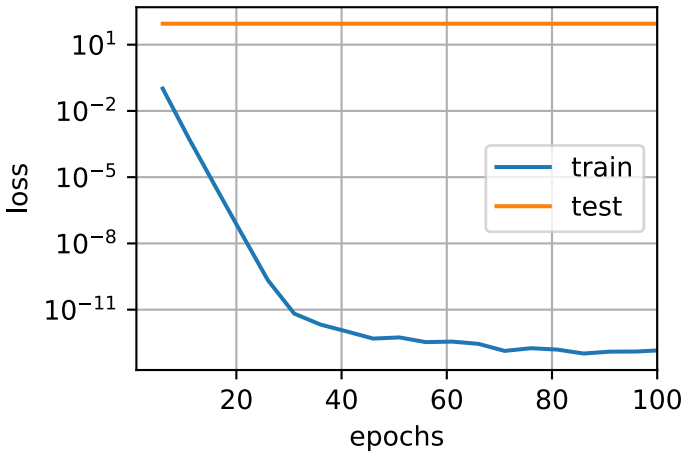
```
def train(lambd):
    w, b = init_params()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    num_epochs, lr = 100, 0.003
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], legend=['train', 'test'])
    for epoch in range(1, num_epochs+1):
        for X, y in train_iter:
            with autograd.record():
                # The L2 norm penalty term has been added
                l = loss(net(X), y) + lambd * l2_penalty(w)
            l.backward()
            d2l.sgd([w, b], lr, batch_size)
        if epoch % 5 == 0:
            animator.add(epoch+1, (d2l.evaluate_loss(net, train_iter, loss),
                                   d2l.evaluate_loss(net, test_iter, loss)))
    print('l1 norm of w:', np.abs(w).sum())
```

Training without Regularization

Next, let us train and test the high-dimensional linear regression model. When `lambd = 0` we do not use weight decay. As a result, while the training error decreases, the test error does not. This is a perfect example of overfitting.

```
train(lambd=0)
```

```
l1 norm of w: 158.557
```

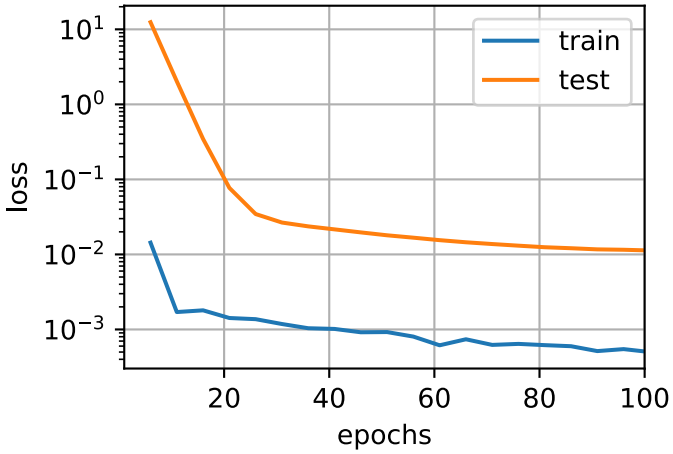


Using Weight Decay

The example below shows that even though the training error increased, the error on the test set decreased. This is precisely the improvement that we expect from using weight decay. While not perfect, overfitting has been mitigated to some extent. In addition, the ℓ_2 norm of the weight $\mathbf{w}$ is smaller than without using weight decay.

```
train(lambd=3)
```

```
l1 norm of w: 0.48914996
```



6.5.4 Concise Implementation

Because weight decay is ubiquitous in neural network optimization, Gluon 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 depdends only on the current value of each parameter, and the optimizer must to touch each parameter once anyway.

In the following code, we specify the weight decay hyperparameter directly through the `wd` parameter when instantiating

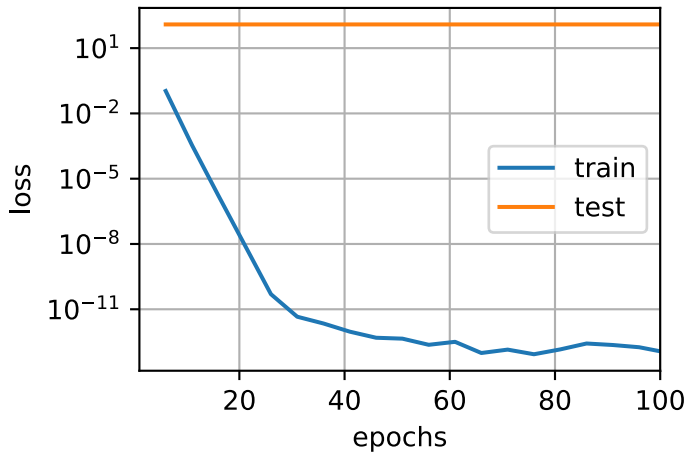
our `Trainer`. By default, Gluon decays both weights and biases simultaneously. Note that we can have *different* optimizers for different sets of parameters. For instance, we can have one `Trainer` with weight decay for the weights $\mathbf{w}$ and another without weight decay to take care of the bias b.

```
def train_gluon(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
    # The weight parameter has been decayed. Weight names generally end with
    # "weight".
    trainer_w = gluon.Trainer(net.collect_params('.*weight'), 'sgd',
                              {'learning_rate': lr, 'wd': wd})
    # The bias parameter has not decayed. Bias names generally end with "bias"
    trainer_b = gluon.Trainer(net.collect_params('.*bias'), 'sgd',
                              {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], legend=['train', 'test'])
    for epoch in range(1, num_epochs+1):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            # Call the step function on each of the two Trainer instances to
            # update the weight and bias separately
            trainer_w.step(batch_size)
            trainer_b.step(batch_size)
        if epoch % 5 == 0:
            animator.add(epoch+1, (d2l.evaluate_loss(net, train_iter, loss),
                                   d2l.evaluate_loss(net, test_iter, loss)))
    print('L1 norm of w:', np.abs(net[0].weight.data()).sum())
```

The plots look just the same as when we implemented weight decay from scratch but they run a bit faster and are easier to implement, a benefit that will become more pronounced for large problems.

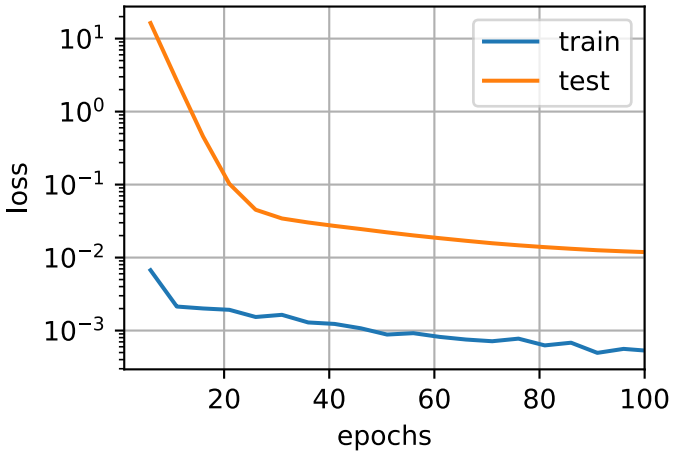
```
train_gluon(0)
```

```
L1 norm of w: 152.12503
```



```
train_gluon(3)
```

```
L1 norm of w: 0.5211031
```



So far, we only touched upon one notion of what constitutes a simple *linear* function. For nonlinear functions, what constitutes *simplicity* can be a far more complex question. For instance, there exist Reproducing Kernel Hilbert Spaces (RKHS)[74] which allow one to use many of the tools introduced for linear functions in a nonlinear context. Unfortunately, RKHS-based algorithms do not always scale well to massive amounts of data. For the purposes of this book, we limit ourselves to simply summing over the weights for different layers, e.g., via $\sum_l \|\mathbf{w}_l\|^2$, which is equivalent to weight decay applied to all layers.

6.5.5 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.
- Gluon provides automatic weight decay functionality in the optimizer by setting the hyperparameter `wd`.
- You can have different optimizers within the same training loop, e.g., for different sets of parameters.

6.5.6 Exercises

1. Experiment with the value of λ in the estimation problem in this page. 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 (this is called ℓ_1 regularization).
4. We know that $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$. Can you find a similar equation for matrices (mathematicians call this the Frobenius norm[75])?

[74] https://en.wikipedia.org/wiki/Reproducing_kernel_Hilbert_space
[75] https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm

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?

6.5.7 Scan the QR Code to Discuss[76]

6.6 Dropout

Just now, we introduced the classical approach of regularizing statistical models by penalyzing 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 0. More intuitively, we might argue that we encouraged the model to spread out its weights among many features and rather than depending too much on a small number of potentially spurious associations.

6.6.1 Overfitting Revisited

Given many more features than examples, linear models can overfit. But when there are many 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: Linear models can't take into account interactions among features. For every feature, a linear model must assign either a positive or a negative weight. They lack the flexibility to account for context.

In more formal texts, you'll see this fundamental tension between generalizability and flexibility discussed as the *bias-variance tradeoff*. Linear models have high bias (they can only represent a small class of functions), but low variance (they give similar results across different random samples of the data).

Deep neural networks take us to the opposite end of the bias-variance spectrum. Neural networks are so flexible because they aren't confined to looking at each feature individually. Instead, 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 "Nigeria" without "Western Union" does not.

Even when we only have a small number of features, deep neural networks are capable of overfitting. In 2017, a group of researchers presented a now well-known demonstration of the incredible flexibility of neural networks. They presented a neural network with randomly-labeled images (there was no true pattern linking the inputs to the outputs) and found that the neural network, optimized by SGD, 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 get better than 10% accuracy on holdout data. Yet even in these situations, when there is no true pattern to be learned, neural networks can perfectly fit the training labels.

[76] https://discuss.mxnet.io/t/2342

6.6.2 Robustness through Perturbations

Let us think briefly about what we expect from a good statistical model. We want it to do well on unseen test data. One way we can accomplish this is by asking what constitutes a a 'simple' model? Simplicity can come in the form of a small number of dimensions, which is what we did when discussing fitting a model with monomial basis functions. Simplicity can also come in the form of a small norm for the basis functions. This led us to weight decay (ℓ_2 regularization). Yet a third notion of simplicity that we can impose is that the function should be robust under small changes in the input. 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 a form of this idea when he proved that training with input noise is equivalent to Tikhonov regularization (Bishop, 1995). In other words, he drew a clear mathematical connection between the requirement that a function be smooth (and thus simple), as we discussed in the section on weight decay, with 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 the 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 deep network with many layers, enforcing smoothness just on the input-output mapping misses out on what is happening internally in the network. Their proposed idea is called *dropout*, and it is now a standard technique that is widely used for training neural networks. Throughout trainin, on each iteration, dropout regularization consists simply of zeroing out some fraction (typically 50%) of the nodes in each layer before calculating the subsequent layer.

The key challenge then is how to inject this noise without introducing undue statistical *bias*. In other words, we want to perturb the inputs to each layer during training in such a way that the expected value of the layer is equal to the value it would have taken had we not introduced any noise at all.

In Bishop's case, when we are adding Gaussian noise to a linear model, this is simple: At each training iteration, just add 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, $\mathbf{E}[\mathbf{x}'] = \mathbf{x}$.

In the case of dropout regularization, one can debias each layer by normalizing by the fraction of nodes that were not dropped out. In other words, dropout with drop probability p is applied as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases} \tag{6.6.1}$$

By design, the expectation remains unchanged, i.e., $\mathbf{E}[h'] = h$. Intermediate activations h are replaced by a random variable h' with matching expectation. The name 'dropout' arises from the notion that some neurons 'drop out' of the computation for the purpose of computing the final result. During training, we replace intermediate activations with random variables.

6.6.3 Dropout in Practice

Recall the multilayer perceptron (Section 6.1) with a hidden layer and 5 hidden units. Its architecture is given by

$$\begin{aligned} h &= \sigma(W_1 x + b_1) \\ o &= W_2 h + b_2 \\ \hat{y} &= \text{softmax}(o) \end{aligned} \tag{6.6.2}$$

When we apply dropout to the hidden layer, we are essentially removing each hidden unit with probability p, (i.e., setting their output to 0). We can view the result as a network containing only a subset of the original neurons. In the image below, h_2 and h_5 are removed. Consequently, the calculation of y no longer depends on h_2 and h_5 and their respective gradient also vanishes when performing backprop. In this way, the calculation of the output layer cannot be overly dependent on any one element of $h_1, \ldots, h_5$. Intuitively, deep learning researchers often explain the intuition thusly: we do not want

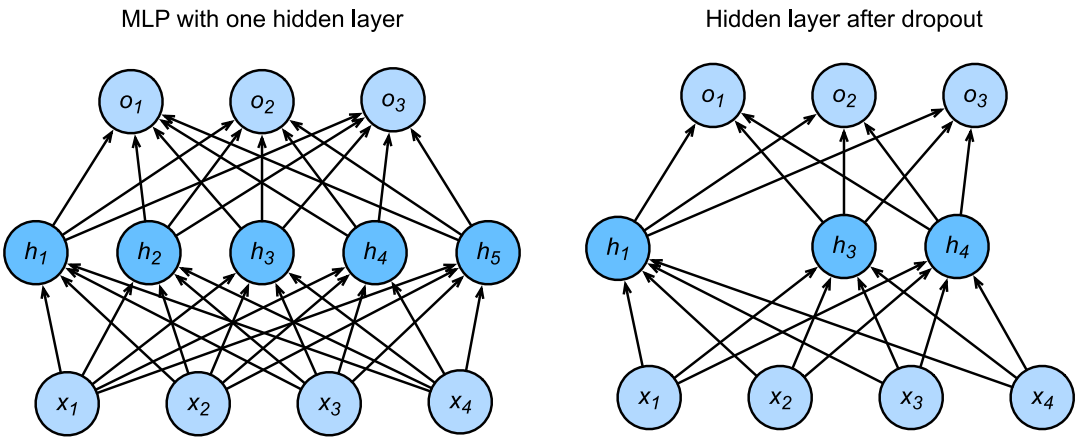


Fig. 6.6.1: MLP before and after dropout

the network's output to depend too precariously on the exact activation pathway through the network. The original authors of the dropout technique described their intuition as an effort to prevent the *co-adaptation* of feature detectors.

At test time, we typically do not use dropout. However, we note that there are some exceptions: some researchers use dropout at test time as a heuristic appraoch for estimating the *confidence* of neural network predictions: if the predictions agree across many different dropout masks, then we might say that the network is more confident. For now we will put off the advanced topic of uncertainty estimation for subsequent chapters and volumes.

6.6.4 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` function that drops out the elements in the `ndarray` input `X` with probability `drop_prob`, rescaling the remainder as described above (dividing the survivors by `1.0-drop_prob`).

```
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

def dropout(X, drop_prob):
    assert 0 <= drop_prob <= 1
    # In this case, all elements are dropped out
    if drop_prob == 1:
        return np.zeros_like(X)
    mask = np.random.uniform(0, 1, X.shape) > drop_prob
    return mask * X / (1.0-drop_prob)
```

We can test out the `dropout` 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(X, 0))
```

(continues on next page)

(continued from previous page)

```
print(dropout(X, 0.5))
print(dropout(X, 1))
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.]
 [ 8.  9. 10. 11. 12. 13. 14. 15.]]
[[ 0.  0.  0.  0.  8. 10. 12.  0.]
 [16.  0. 20. 22.  0.  0.  0. 30.]]
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

Defining Model Parameters

Again, we can use the Fashion-MNIST dataset, introduced in Section 5.6. We will define a multilayer perceptron with two hidden layers. The two hidden layers both have 256 outputs.

```
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()
```

Define the Model

The model defined below concatenates the fully-connected layer and the activation function ReLU, using dropout for the output of each activation function. We can set the dropout probability of each layer separately. It is generally recommended 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 layer respectively. By using the `is_training` function described in Section 4.6, we can ensure that dropout is only active during training.

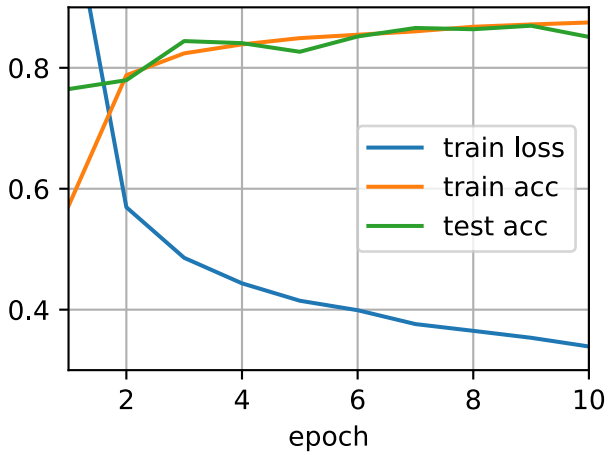
```
drop_prob1, drop_prob2 = 0.2, 0.5

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(H1, drop_prob1)
    H2 = npx.relu(np.dot(H1, W2) + b2)
    if autograd.is_training():
        # Add a dropout layer after the second fully connected layer
        H2 = dropout(H2, drop_prob2)
    return np.dot(H2, W3) + b3
```

Training and Testing

This is similar to the training and testing of multilayer perceptrons 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))
```



6.6.5 Concise Implementation

Using Gluon, all we need to do is add a `Dropout` layer (also in the `nn` package) 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 subequent layer) according to the specified dropout probability. When MXNet is not in training mode, the `Dropout` layer simply passes the data through during testing.

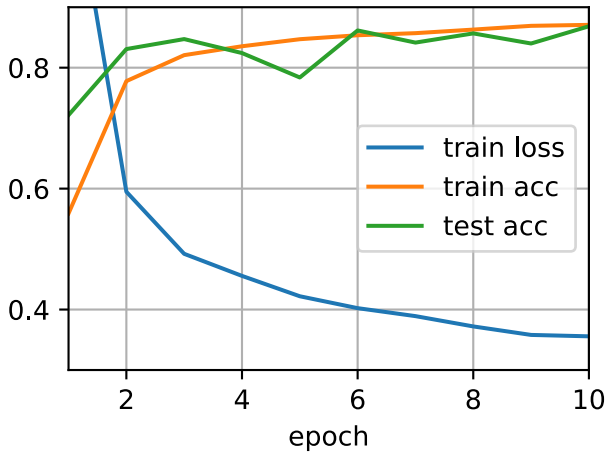
```
net = nn.Sequential()
net.add(nn.Dense(256, activation="relu"),
        # Add a dropout layer after the first fully connected layer
        nn.Dropout(drop_prob1),
        nn.Dense(256, activation="relu"),
        # Add a dropout layer after the second fully connected layer
        nn.Dropout(drop_prob2),
        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)
```

6.6.6 Summary

- Beyond controlling the number of dimensions and the size of the weight vector, dropout is yet another tool to avoid overfitting. Often all three are used jointly.



- Dropout replaces an activation h with a random variable h' with expected value h and with variance given by the dropout probability p.
- Dropout is only used during training.

6.6.7 Exercises

1. Try out what happens if you change the dropout probabilities for layers 1 and 2. In particular, what happens if you switch the ones for both layers?
2. Increase the number of epochs and compare the results obtained when using dropout with those when not using it.
3. Compute the variance of the the activation random variables after applying dropout.
4. Why should you typically not using dropout?
5. If changes are made to the model to make it more complex, such as adding hidden layer units, will the effect of using dropout to cope with overfitting be more obvious?
6. Using the model in this section as an example, compare the effects of using dropout and weight decay. What if dropout and weight decay are used at the same time?
7. What happens if we apply dropout to the individual weights of the weight matrix rather than the activations?
8. Replace the dropout activation with a random variable that takes on values of $[0, \gamma/2, \gamma]$. Can you design something that works better than the binary dropout function? Why might you want to use it? Why not?

6.6.8 Scan the QR Code to Discuss[77]

[77] https://discuss.mxnet.io/t/2343

6.7 Forward Propagation, Backward Propagation, and Computational Graphs

In the previous sections, we used minibatch stochastic gradient descent to train our models. When we implemented the algorithm, we only worried about the calculations involved in *forward propagation* through the model. In other words, we implemented the calculations required for the model to generate output corresponding to come given input, but when it came time to calculate the gradients of each of our parameters, we invoked the `backward` function, relying on the `autograd` module to figure out what to do.

The automatic calculation of gradients profoundly simplifies the implementation of deep learning algorithms. Before automatic differentiation, even small changes to complicated models would require recalculating lots of derivatives by hand. Even academic papers would too often have to allocate lots of page real estate to deriving update rules.

While we plan to continue relying on `autograd`, and we have already come a long way without every discussing how these gradients are calculated efficiently under the hood, it is important that you know how updates are actually calculated if you want to go beyond a shallow understanding of deep learning.

In this section, we will peel back the curtain on some of the details of backward propagation (more commonly called *backpropagation* or *backprop*). To convey some insight for both the techniques and how they are implementated, we will rely on both mathematics and computational graphs to describe the mechanics behind neural network computations. To start, we will focus our exposition on a simple multilayer perceptron with a single hidden layer and ℓ_2 norm regularization.

6.7.1 Forward Propagation

Forward propagation refers to the calculation and storage of intermediate variables (including outputs) for the neural network within the models in the order from input layer to output layer. In the following, we work in detail through the example of a deep network with one hidden layer step by step. This is a bit tedious but it will serve us well when discussing what really goes on when we call `backward`.

For the sake of simplicity, let's assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and there is no bias term. Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} \tag{6.7.1}$$

$\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer. After entering the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ into the activation function ϕ operated by the basic elements, we will obtain a hidden layer variable with the vector length of h,

$$\mathbf{h} = \phi(\mathbf{z}). \tag{6.7.2}$$

The hidden variable $\mathbf{h}$ is also an intermediate variable. Assuming 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 length of q:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \tag{6.7.3}$$

Assuming the loss function is l and the example label is y, we can then calculate the loss term for a single data example,

$$L = l(\mathbf{o}, y). \tag{6.7.4}$$

According to the definition of ℓ_2 norm regularization, given the hyperparameter λ, the regularization term is

$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right), \tag{6.7.5}$$

where the Frobenius norm of the matrix is equivalent to the calculation of the L_2 norm after flattening the matrix to a vector. Finally, the model's regularized loss on a given data example is

$$J = L + s. \tag{6.7.6}$$

We refer to J as the objective function of a given data example and refer to it as the 'objective function' in the following discussion.

6.7.2 Computational Graph of Forward Propagation

Plotting computational graphs helps us visualize the dependencies of operators and variables within the calculation. The figure below contains the graph associated with the simple network described above. The lower-left corner signifies the input and the upper right corner the output. Notice that the direction of the arrows (which illustrate data flow) are primarily rightward and upward.

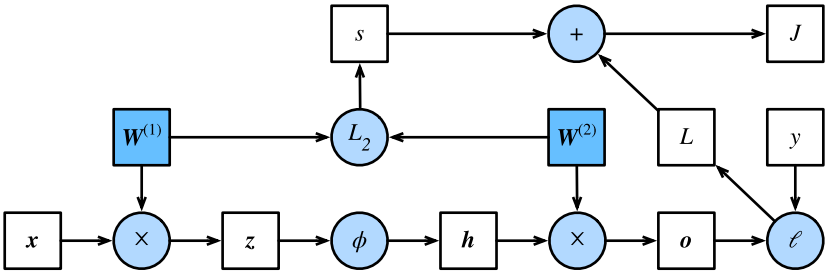


Fig. 6.7.1: Computational Graph

6.7.3 Backpropagation

Backpropagation refers to the method of calculating the gradient of neural network parameters. In general, back propagation calculates and stores the intermediate variables of an objective function related to each layer of the neural network and the gradient of the parameters in the order of the output layer to the input layer according to the 'chain rule' in calculus. Assume that we have functions $\mathsf{Y} = f(\mathsf{X})$ and $\mathsf{Z} = g(\mathsf{Y}) = g \circ f(\mathsf{X})$, 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 wrt. X via

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}}\right). \tag{6.7.7}$$

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 and for higher dimensional tensors we use the appropriate counterpart. The operator prod hides all the notation overhead.

The parameters of the simple network with one hidden layer 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 accompish this, we will 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 compute 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.

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1 \tag{6.7.8}$$

Next, we compute the gradient of the objective function with respect to variable of the output layer $\mathbf{o}$ according to the chain rule.

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q \tag{6.7.9}$$

Next, we calculate the gradients of the regularization term with respect to both parameters.

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)} \tag{6.7.10}$$

Now we are able 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:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)} \tag{6.7.11}$$

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

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}. \tag{6.7.12}$$

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 $\odot$.

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right). \tag{6.7.13}$$

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

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \tag{6.7.14}$$

6.7.4 Training a Model

When training networks, forward and backward propagation depend on each other. In particular, for forward propagation, we traverse the compute 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. 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 backpropagation requires significantly more memory than plain 'inference'—we end up computing tensors as gradients and need to retain all the intermediate variables to invoke the chain rule. Another reason is that we typically train with minibatches containing more than one variable, thus more intermediate activations need to be stored.

6.7.5 Summary

- Forward propagation sequentially calculates and stores intermediate variables within the compute graph defined by the neural network. It proceeds from input to output layer.
- Back propagation 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 and storage.

6.7.6 Exercises

1. Assume that the inputs $\mathbf{x}$ are matrices. What is the dimensionality of the gradients?
2. Add a bias to the hidden layer of the model described in this chapter.
    - Draw the corresponding compute graph.
    - Derive the forward and backward propagation equations.
3. Compute the memory footprint for training and inference in model described in the current chapter.
4. Assume that you want to compute *second* derivatives. What happens to the compute graph? Is this a good idea?
5. Assume that the compute graph is too large for your GPU.
    - Can you partition it over more than one GPU?
    - What are the advantages and disadvantages over training on a smaller minibatch?

6.7.7 Scan the QR Code to Discuss[78]

6.8 Numerical Stability and Initialization

In the past few sections, each model that we implemented required initializing our parameters according to some specified distribution. However, until now, we glossed over the details, taking the initialization hyperparameters for granted. You might even have gotten the impression that these choices are not especially important. However, the choice of initialization scheme plays a significant role in neural network learning, and can prove essentially to maintaining numerical stability. Moreover, these choices can be tied up in interesting ways with the choice of the activation function. Which nonlinear activation function we choose, and how we decide to initialize our parameters can play a crucial role in making the optimization algorithm converge rapidly. Failure to be mindful of these issues can lead to either exploding or vanishing gradients. In this section, we delve into these topics with greater detail and discuss some useful heuristics that you may use frequently throughout your career in deep learning.

6.8.1 Vanishing and Exploding Gradients

Consider a deep network with d layers, input $\mathbf{x}$ and output $\mathbf{o}$. Each layer satisfies:

$$\mathbf{h}^{t+1} = f_t(\mathbf{h}^t) \text{ and thus } \mathbf{o} = f_d \circ \ldots, \circ f_1(\mathbf{x}) \tag{6.8.1}$$

If all activations and inputs are vectors, we can write the gradient of $\mathbf{o}$ with respect to any set of parameters $\mathbf{W}_t$ associated with the function f_t at layer t simply as

$$\partial_{\mathbf{W}_t} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{d-1}} \mathbf{h}^d}_{:=\mathbf{M}_d} \cdot \ldots, \cdot \underbrace{\partial_{\mathbf{h}^t} \mathbf{h}^{t+1}}_{:=\mathbf{M}_t} \underbrace{\partial_{\mathbf{W}_t} \mathbf{h}^t}_{:=\mathbf{v}_t}. \tag{6.8.2}$$

In other words, it is the product of $d-t$ matrices $\mathbf{M}_d \cdot \ldots, \cdot \mathbf{M}_t$ and the gradient vector $\mathbf{v}_t$. What happens is similar to the situation when we experienced numerical underflow when multiplying too many probabilities. At the time, we were able to mitigate the problem by switching from into log-space, i.e., by shifting the problem from the mantissa to the exponent of the numerical representation. Unfortunately the problem outlined in the equation above is much more serious: initially the matrices M_t may well have a wide variety of eigenvalues. They might be small, they might be large, and in particular, their product might well be *very large* or *very small*. This is not (only) a problem of numerical representation but it means that the optimization algorithm is bound to fail. It receives gradients that are either excessively large or excessively small. As a result the steps taken are either (i) excessively large (the *exploding* gradient problem), in which case the parameters blow up in magnitude rendering the model useless, or (ii) excessively small, (the *vanishing gradient problem*), in which case the parameters hardly move at all, and thus the learning process makes no progress.

Vanishing Gradients

One major culprit in the vanishing gradient problem is the choices of the activation functions σ that are interleaved with the linear operations in each layer. Historically, the sigmoid function $(1 + \exp(-x))$ (introduced in Section 6.1) was a popular choice owing to its similarity to a thresholding function. Since early artificial neural networks were inspired by biological neural networks, the idea of neurons that either fire or do not fire (biological neurons do not partially fire)

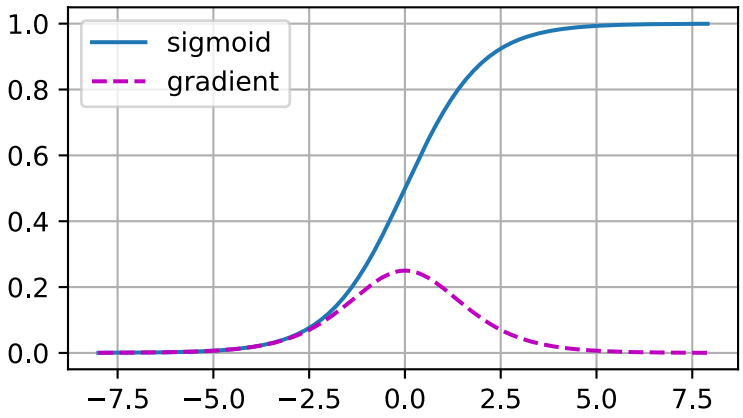
[78] https://discuss.mxnet.io/t/2344

seemed appealing. Let us take a closer look at the function to see why picking it might be problematic vis-a-vis vanishing gradients.

```
%matplotlib inline
import d2l
from mxnet import np, npx, autograd
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 we can see, the gradient of the sigmoid vanishes both when its inputs are large and when they are small. Moreover, when we excecute backward propagation, due to the chain rule, this means that unless we are in the Goldilocks zone, where the inputs to most of the sigmoids are in the range of, say $[-4, 4]$, the gradients of the overall product may vanish. When we have many layers, unless we are especially careful, we are likely to find that our gradient is cut off at *some* layer. Before ReLUs ($\max(0, x)$) were proposed as an alternative to squashing functions, this problem used to plague deep network training. As a consequence, ReLUs have become the default choice when designing activation functions in deep networks.

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. If this were to happen to us with a deep network, we would have no realistic 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   0.7740038   1.0434405   1.1839255 ]
 [ 1.8917114  -1.2347414  -1.771029   -0.45138445]
 [ 0.57938355 -1.856082   -1.9768796  -0.20801921]
 [ 0.2444218  -0.03716067 -0.48774993 -0.02261727]]
After multiplying 100 matrices [[ 3.1575275e+20 -5.0052276e+19  2.0565092e+21 -2.
↪3741922e+20]
 [-4.6332600e+20  7.3445046e+19 -3.0176513e+21  3.4838066e+20]
 [-5.8487235e+20  9.2711797e+19 -3.8092853e+21  4.3977330e+20]
 [-6.2947415e+19  9.9783660e+18 -4.0997977e+20  4.7331174e+19]]
```

Symmetry

Another problem in deep network design is the symmetry inherent in their parametrization. Assume that we have a deep network with one hidden layer with two units, say h_1 and h_2. 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. Imagine what would happen if we initialized all of the parameters of some layer as $\mathbf{W}_l = c$ for some constant c. In this case, the gradients for all dimensions are identical: thus not only would each unit take the same value, but it would receive the same update. Stochastic gradient descent would never break the symmetry on its own and we might never be able to realize the networks expressive power. The hidden layer would behave as if it had only a single unit. As an aside, note that while SGD would not break this symmetry, dropout regularization would!

6.8.2 Parameter Initialization

One way of addressing, or at least mitigating the issues raised above is through careful initialization of the weight vectors. This way we can ensure that (at least initially) the gradients do not vanish and that they maintain a reasonable scale where the network weights do not diverge. Additional care during optimization and suitable regularization ensures that things never get too bad.

Default Initialization

In the previous sections, e.g., in Section 5.3, we used `net.initialize(init.Normal(sigma=0.01))` to initialize the values of our weights. If the initialization method is not specified, such as `net.initialize()`, MXNet will use the default random initialization method: each element of the weight parameter is randomly sampled with a uniform distribution $U[-0.07, 0.07]$ and the bias parameters are all set to 0. Both choices tend to work well in practice for moderate problem sizes.

Xavier Initialization

Let us look at the scale distribution of the activations of the hidden units h_i for some layer. They are given by

$$h_i = \sum_{j=1}^{n_{\text{in}}} W_{ij} x_j \tag{6.8.3}$$

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 (this does not mean that the distribution has to be Gaussian, just that mean and variance need to exist). We do not really have much control over the inputs into the layer x_j but let us proceed with the somewhat

unrealistic assumption that they also have zero mean and variance γ^2 and that they are independent of $\mathbf{W}$. In this case, we can compute mean and variance of h_i as follows:

$$\begin{aligned}\mathbf{E}[h_i] &= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij} x_j] = 0 \\ \mathbf{E}[h_i^2] &= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij}^2 x_j^2] \\ &= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij}^2] \mathbf{E}[x_j^2] \\ &= n_{\text{in}} \sigma^2 \gamma^2\end{aligned} \tag{6.8.4}$$

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 top layers. That is, instead of $\mathbf{Ww}$, we need to deal with $\mathbf{W}^\top \mathbf{g}$, where $\mathbf{g}$ is the incoming gradient from the layer above. Using the same reasoning as for forward propagation, we see that the gradients' variance can blow up unless $n_{\text{out}}\sigma^2 = 1$. This leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously. Instead, we simply try to satisfy:

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}. \tag{6.8.5}$$

This is the reasoning underlying the eponymous Xavier initialization (Glorot & Bengio, 2010). It works well enough in practice. For Gaussian random variables, the Xavier initialization picks a normal distribution with zero mean and variance $\sigma^2 = 2/(n_{\text{in}} + n_{\text{out}})$. For uniformly distributed random variables $U[-a, a]$, note that their variance is given by $a^2/3$. Plugging $a^2/3$ into the condition on σ^2 yields that we should initialize uniformly with $U\left[-\sqrt{6/(n_{\text{in}} + n_{\text{out}})}, \sqrt{6/(n_{\text{in}} + n_{\text{out}})}\right]$.

Beyond

The reasoning above barely scratches the surface of modern approaches to parameter initialization. In fact, MXNet has an entire `mxnet.initializer` module implementing over a dozen different heuristics. Moreover, intialization continues to be a hot area of inquiry within research into the fundamental theory of neural network optimization. Some of these heuristics are especially suited for when parameters are tied (i.e., when parameters of in different parts the network are shared), for superresolution, sequence models, and related problems. We recommend that the interested reader take a closer look at what is offered as part of this module, and investigate the recent research on parameter initialization. Perhaps you may come across a recent clever idea and contribute its implementation to MXNet, or you may even invent your own scheme!

6.8.3 Summary

- Vanishing and exploding gradients are common issues in very deep networks, unless great care is taking to ensure that gradients and parameters remain well controlled.
- Initialization heuristics are needed to ensure that at least the initial gradients are neither too large nor too small.
- The ReLU addresses one of the vanishing gradient problems, namely that gradients vanish for very large inputs. This can accelerate convergence significantly.
- Random initialization is key to ensure that symmetry is broken before optimization.

6.8.4 Exercises

1. Can you design other cases of symmetry breaking besides the permutation symmetry?
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 LARS by You, Gitman and Ginsburg, 2017[79] for inspiration.

6.8.5 Scan the QR Code to Discuss[80]

6.9 Considering the Environment

So far, we have worked through a number of hands-on implementations fitting machine learning models to a variety of datasets. And yet, until now we skated over the matter of where are data comes from in the first place, and what we plan to ultimately *do* with the outputs from our models. Too often in the practice of machine learning, developers rush ahead with the development of models tossing these fundamental considerations aside.

Many failed machine learning deployments can be traced back to this situation. Sometimes the model does well as evaluated by test accuracy only to fail catastrophically in the real world when the distribution of data suddenly shifts. More insidiously, sometimes the very deployment of a model can be the catalyst which perturbs the data distribution. Say for example that we trained a model to predict loan defaults, finding that the choice of footware was associated with 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. But our ill-conceived leap from pattern recognition to decision-making and our failure to think critically about the environment might have disastrous consequences. For starters, as soon as we began making decisions based on footware, customers would catch on and change their behavior. Before long, all applicants would be wearing Oxfords, and yet there would be no coinciding improvement in credit-worthiness. Think about this deeply because similar issues abound in the application of machine learning: by introducing our model-based decisions to the environment, we might break the model.

In this chapter, we describe some common concerns and aim to get you started acquiring the critical thinking that you will need in order to detect these situations early, mitigate the 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 entre the realm of philosophy and grapple with difficult questions concerning ethics and informed consent.

6.9.1 Distribution Shift

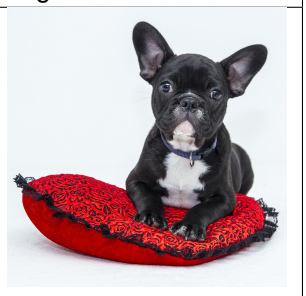
To begin, we return to the observational setting, putting aside for now the impacts of our actions on the environment. In the following sections, we take a deeper look at the various ways that data distributions might shift, and what might be done to salvage model performance. From the outset, we should warn that if the data-generating distribution $p(\mathbf{x}, y)$ can

[79] https://arxiv.org/pdf/1708.03888.pdf
[80] https://discuss.mxnet.io/t/2345

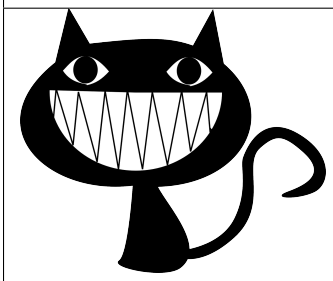
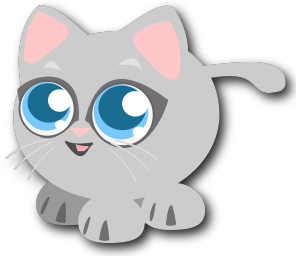
shift in arbitrary ways at any point in time, then learning a robust classifier is impossible. In the most pathological case, if the label definitions themselves can change at a moments notice: if suddenly what we called "cats" are now dogs and what we previously called "dogs" are now in fact cats, without any perceptible change in the distribution of inputs $p(\mathbf{x})$, then there is nothing we could do to detect the change or to correct our classifier at test time. Fortunately, under some restricted assumptions on the ways our data might change in the future, principled algorithms can detect shift and possibly even adapt, achieving higher accuracy than if we naively continued to rely on our original classifier.

Covariate Shift

One of the best-studied forms of distribution shift is *covariate shift*. Here we assume that although the distribution of inputs may change over time, the labeling function, i.e., the conditional distribution $P(y \mid \mathbf{x})$ does not change. While this problem is easy to understand its also easy to overlook it in practice. Consider the challenge of distinguishing cats and dogs. Our training data consists of images of the following kind:

cat	cat	dog	dog
			

At test time we are asked to classify the following images:

cat	cat	dog	dog
			

Obviously this is unlikely to work well. The training set consists of photos, while the test set contains only cartoons. The colors are not even realistic. Training on a dataset that looks substantially different from the test set without some plan for how to adapt to the new domain is a bad idea. Unfortunately, this is a very common pitfall. Statisticians call this *covariate shift* because the root of the problem owed to a shift in the distribution of features (i.e., of *covariates*). Mathematically, we could say that $P(\mathbf{x})$ changes but that $P(y \mid \mathbf{x})$ remains unchanged. Although its usefulness is not restricted to this setting, when we believe $\mathbf{x}$ causes y, covariate shift is usually the right assumption to be working with.

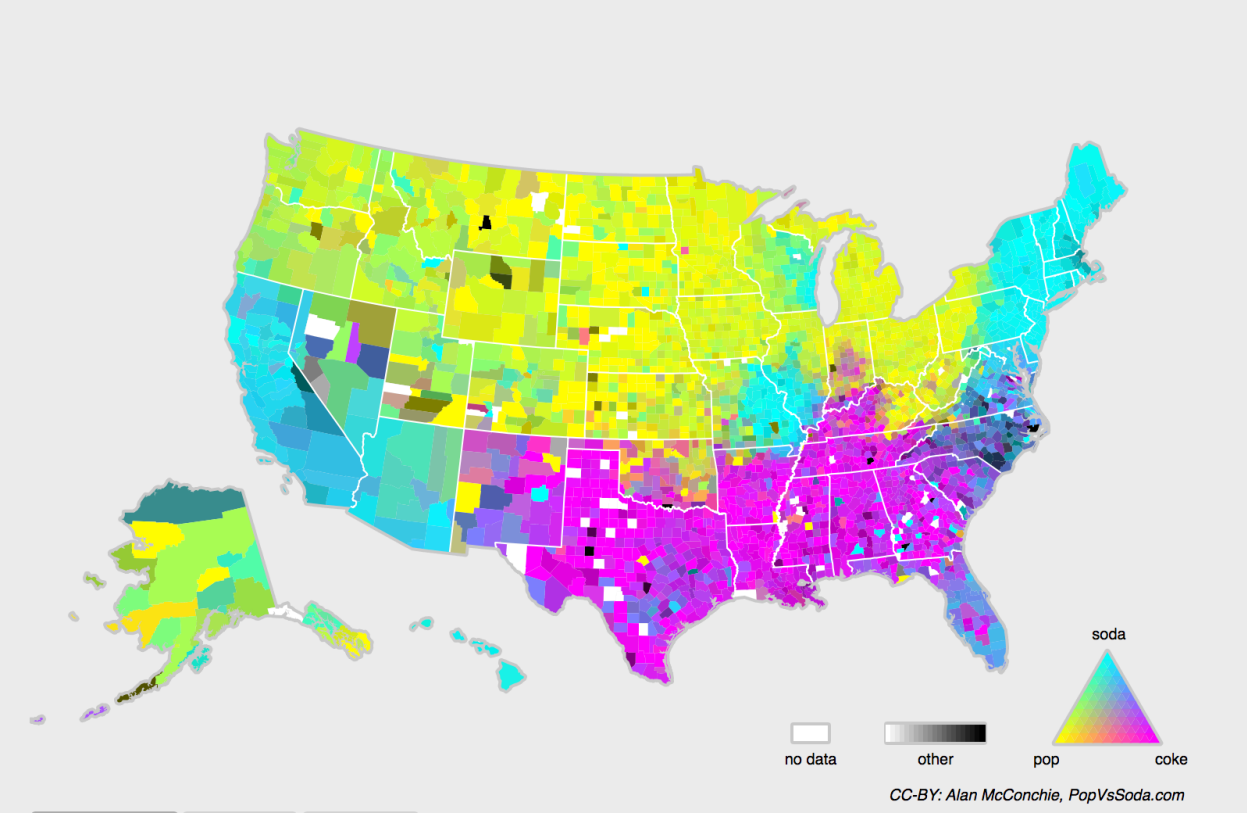
Label Shift

The converse problem emerges when we believe that what drives the shift is a change in the marginal distribution over the labels $P(y)$ but that the class-conditional distributions are invariant $P(\mathbf{x} \mid y)$. Label shift is a reasonable assumption to make when we believe that y causes $\mathbf{x}$. For example, commonly we want to predict a diagnosis given its manifestations. In this case we believe that the diagnosis causes the manifestations, i.e., diseases cause symptoms. Sometimes the label shift and covariate shift assumptions can hold simultaneously. For example, when the true labeling function is deterministic

and unchanging, then covariate shift will always hold, including if label shift holds too. Interestingly, when we expect both label shift and covariate shift hold, it is often advantageous to work with the methods that flow from the label shift assumption. That is because these methods tend to involve manipulating objects that look like the label, which (in deep learning) tends to be comparatively easy compared to working with the objects that look like the input, which tends (in deep learning) to be a high-dimensional object.

Concept Shift

One more related problem arises in *concept shift*, the situation in which the very label definitions change. This sounds weird—after all, a *cat* is a *cat*. Indeed the definition of a cat might not change, but can we say the same about soft drinks? 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 definition of even this simple term:



If we were to build a machine translation system, the distribution $P(y \mid x)$ might be different depending on our location. This problem can be tricky to spot. A saving grace is that often the $P(y \mid x)$ only shifts gradually. Before we go into further detail and discuss remedies, we can discuss a number of situations where covariate and concept shift may not be so obvious.

Examples

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're ready for a successful career in medical diagnostics. Not so fast…

Many things could go wrong. In particular, the distributions that you work with for training and those that you encounter in the wild might differ considerably. This happened to an unfortunate startup, that Alex had the opportunity to consult for many years ago. They were developing a blood test for a disease that affects mainly older men and they'd managed to obtain a fair amount of blood samples from patients. It is considerably more difficult, though, to obtain blood samples from healthy men (mainly for ethical reasons). To compensate for that, they asked a large number of students on campus to donate blood and they performed their test. Then they asked me whether I could help them build a classifier to detect the disease. I told them that it would be very easy to distinguish between both datasets with near-perfect accuracy. After all, 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: Their sampling procedure made it likely that an extreme case of covariate shift would arise between the *source* and *target* distributions, and at that, one that could not be corrected by conventional means. In other words, training and test data were so different that nothing useful could be done and they had wasted significant amounts of money.

Self Driving Cars

Say a company wanted to build a machine learning system for self-driving cars. One of the key components is a roadside detector. Since real annotated data is expensive to get, they had the (smart and questionable) idea to use synthetic data from a game rendering engine as additional training data. This worked really well on 'test data' drawn from the rendering engine. Alas, inside a real car it was a disaster. As it turned out, the roadside had been rendered with a very simplistic texture. More importantly, *all* the roadside had been rendered with the *same* texture and the roadside detector learned about this 'feature' very quickly.

A similar thing happened to the US Army when they first tried to detect tanks in the forest. They took aerial photographs of the forest without tanks, then drove the tanks into the forest and took another set of pictures. The so-trained classifier worked 'perfectly'. Unfortunately, all it had learned was to distinguish trees with shadows from trees without shadows—the first set of pictures was taken in the early morning, the second one at noon.

Nonstationary distributions

A much more subtle situation arises when the distribution changes slowly and the model is not updated adequately. Here 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 it keeps on recommending Santa hats long after Christmas.

More Anecdotes

- We build a face detector. It works well on all benchmarks. Unfortunately it fails on test data - the offending examples are close-ups where the face fills the entire image (no such data was in the training set).
- We build a web search engine for the USA 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.

In short, 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.

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 x_i are drawn from some *target* distribution $q(\mathbf{x})$ rather than the *source* distribution $p(\mathbf{x})$. To make progress, we need to reflect about what exactly is happening during training: we iterate over training data and associated labels $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and update the weight vectors of the model after every minibatch. We sometimes additionally apply some penalty to the parameters, using weight decay, dropout, or some other related technique. This means that we largely minimize the loss on the training.

$$\underset{w}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(x_i, y_i, f(x_i)) + \text{some penalty}(w) \tag{6.9.1}$$

Statisticians call the first term an *empirical average*, i.e., an average computed over the data drawn from $P(x)P(y \mid x)$. If the data is drawn from the 'wrong' distribution q, we can correct for that by using the following simple identity:

$$\begin{aligned} \int p(\mathbf{x}) f(\mathbf{x}) dx &= \int p(\mathbf{x}) f(\mathbf{x}) \frac{q(\mathbf{x})}{p(\mathbf{x})} dx \\ &= \int q(\mathbf{x}) f(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} dx \end{aligned} \tag{6.9.2}$$

In other words, we need to re-weight each instance by the ratio of probabilities that it would have been drawn from the correct distribution $\beta(\mathbf{x}) := p(\mathbf{x})/q(\mathbf{x})$. 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 training data, and the one used for generating the training set q (the latter is trivially available). Note however, that we only need samples $\mathbf{x} \sim q(\mathbf{x})$; we do not to access labels $y \sim q(y)$.

In this case, there exists a very effective approach that will give almost as good results: logistic regression. This is all that is needed to compute estimate probability ratios. We learn a classifier to distinguish between data drawn from $p(\mathbf{x})$ and data drawn from $q(x)$. 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 over/underweighted accordingly. For simplicity's sake assume that we have an equal number of instances from both distributions, denoted by $\mathbf{x}_i \sim p(\mathbf{x})$ and $\mathbf{x}'_i \sim q(\mathbf{x})$ respectively. Now denote by z_i labels which are 1 for data drawn from p and -1 for data drawn from q. Then the probability in a mixed dataset is given by

$$P(z = 1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x}) + q(\mathbf{x})} \text{ and hence } \frac{P(z = 1 \mid \mathbf{x})}{P(z = -1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})} \tag{6.9.3}$$

Hence, if we use a logistic regression approach where $P(z = 1 \mid \mathbf{x}) = \frac{1}{1+\exp(-f(\mathbf{x}))}$ it follows that

$$\beta(\mathbf{x}) = \frac{1/(1 + \exp(-f(\mathbf{x})))}{\exp(-f(\mathbf{x})/(1 + \exp(-f(\mathbf{x}))))} = \exp(f(\mathbf{x})) \tag{6.9.4}$$

As a result, we need to solve two problems: first one to distinguish between data drawn from both distributions, and then a reweighted minimization problem where we weigh terms by β, e.g., via the head gradients. Here's a prototypical algorithm for that purpose which uses an unlabeled training set X and test set Z:

1. Generate training set with $\{(\mathbf{x}_i, -1) ... (\mathbf{z}_j, 1)\}$
2. Train binary classifier using logistic regression to get function f
3. Weigh training data using $\beta_i = \exp(f(\mathbf{x}_i))$ or better $\beta_i = \min(\exp(f(\mathbf{x}_i)), c)$

4. Use weights β_i for training on X with labels Y

Note that this method relies on a crucial assumption. For this scheme to work, we need that each data point in the target (test time)distribution had nonzero probability of occurring at training time. If we find a point where $q(\mathbf{x}) > 0$ but $p(\mathbf{x}) = 0$, then the corresponding importance weight should be infinity.

Generative Adversarial Networks use a very similar idea to that described above to engineer a *data generator* that outputs data that cannot be distinguished from examples sampled from a reference dataset. In these approaches, we use one network, f to distinguish real versus fake data and a second network g that tries to fool the discriminator f into accepting fake data as real. We will discuss this in much more detail later.

Label Shift Correction

For the discussion of label shift, we will assume for now that we are dealing with a k-way multiclass classification task. When the distribution of labels shifts over time $p(y) \neq q(y)$ but the class-conditional distributions stay the same $p(\mathbf{x}) = q(\mathbf{x})$, our importance weights will correspond to the label likelihood ratios $q(y)/p(y)$. 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 are often high-dimensional perceptual objects like images, while the labels are often easier to work, say vectors whose length corresponds to the number of classes).

To estimate calculate 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 C, is simply a $k \times k$ matrix where each column corresponds to the *actual* label and each row corresponds to our model's predicted label. Each cell's value c_{ij} is the fraction of predictions where the true label was j *and* our model predicted y.

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 output μ_y.

It turns out that under some mild conditions— if our classifier was reasonably accurate in the first place, if the target data contains only classes of images that we have seen before, and if the label shift assumption holds in the first place (far the strongest assumption here), then we can recover the test set label distribution by solving a simple linear system $C \cdot q(y) = \mu_y$. If our classifier is sufficiently accurate to begin with, then the confusion C will be invertible, and we get a solution $q(y) = C^{-1}\mu_y$. Here we abuse notation a bit, using $q(y)$ to denote the vector of label frequencies. Because we observe the labels on the source data, it is easy to estimate the distribution $p(y)$. Then for any training example i with label y, we can take the ratio of our estimates $\hat{q}(y)/\hat{p}(y)$ to calculate the weight w_i, and plug this into the weighted risk minimization algorithm above.

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 cameras 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.

6.9.2 A Taxonomy of Learning Problems

Armed with knowledge about how to deal with changes in $p(x)$ and in $P(y \mid x)$, we can now consider some other aspects of machine learning problems formulation.

- **Batch Learning.** Here we have access to training data and labels $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, which we use to train a network $f(x, w)$. Later on, we deploy this network to score new data (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 (x_i, y_i) arrives one sample at a time. More specifically, assume that we first observe x_i, then we need to come up with an estimate $f(x_i, w)$ 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. E.g. 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, we have the following cycle where we are continuously improving our model given new observations.

$$\text{model } f_t \longrightarrow \text{data } x_t \longrightarrow \text{estimate } f_t(x_t) \longrightarrow \text{observation } y_t \longrightarrow \text{loss } l(y_t, f_t(x_t)) \longrightarrow \text{model } f_{t+1} \tag{6.9.5}$$

- **Bandits.** They 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 (and nonadversarial Reinforcement Learning).** In many cases the environment remembers what we did. Not necessarily in an adversarial manner but it'll just remember and the response will depend on what happened before. E.g. 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 (i.e., to reduce variance).
- **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. 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, trying to cooperate with it, etc.

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.

6.9.3 Fairness, Accountability, and Transparency in machine Learning

Finally, it is important to remember that when you deploy machine learning systems you are not simply minimizing negative log likelihood or maximizing accuracy—you are automating some kind of decision process. Often the automated decision-making systems that we deploy can have consequences for those subject to its decisions. 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 would run afoul of basic ethical principles. Moreover, "accuracy" is seldom the right metric. When translating predictions in to actions we will often want to take into account the potential cost sensitivity of erring in various ways. If one way that you might classify an image could be perceived as a racial sleight, while misclassification to a different category would be harmless, then you might want to adjust your 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, if prediction systems are applied naively to predictive policing, allocating patrol officers accordingly, a vicious cycle might emerge. Neighborhoods that have more crimes, get more patrols, get more crimes discovered, get more training data, get yet more confident predictions, leading to even more patrols, even more crimes discovered, etc. 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 what news someone is exposed to be determined by which Facebook pages they have *Liked*? These are just a few among the many profound ethical dilemmas that you might encounter in a career in machine learning.

6.9.4 Summary

- In many cases training and test set do not come from the same distribution. This is called covariate shift.
- Covariate shift can be detected and corrected if the shift is not too severe. Failure to do so leads to nasty surprises at test time.
- In some cases the environment *remembers* what we did and will respond in unexpected ways. We need to account for that when building models.

6.9.5 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. What could go wrong if training and test set are very different? What would happen to the sample weights?

6.9.6 Scan the QR Code to Discuss[81]

[81] https://discuss.mxnet.io/t/2347

6.10 Predicting House Prices on Kaggle

In the previous sections, we introduced the basic tools for building deep networks and performing capacity control via dimensionality-reduction, weight decay and dropout. You are now ready to put all this knowledge into practice by participating in a Kaggle competition. Predicting house prices[82] is a great place to start: the data is reasonably generic and does not have the kind of rigid structure that might require specialized models the way images or audio might. This dataset, collected by Bart de Cock[83] in 2011, is considerably larger than the famous the Boston housing dataset[84] of Harrison and Rubinfeld (1978). It boasts both more examples and more features, covering house prices in Ames, IA from the period of 2006-2010.

In this section, we will walk you through details of data preprocessing, model design, hyperparameter selection and tuning. We hope that through a hands-on approach, you will be able to observe the effects of capacity control, feature extraction, etc. in practice. This experience is vital to gaining intuition as a data scientist.

6.10.1 Kaggle

Kaggle[85] is a popular platform for machine learning competitions. It combines data, code and users in a way to allow for both collaboration and competition. While leaderboard chasing can sometimes get out of control, there is also a lot to be said for the objectivity in a platform that provides fair and direct quantitative comparisons between your approaches and those devised by your competitors. Moreover, you can checkout the code from (some) other competitors' submissions and pick apart their methods to learn new techniques. If you want to participate in one of the competitions, you need to register for an account (do this now!).

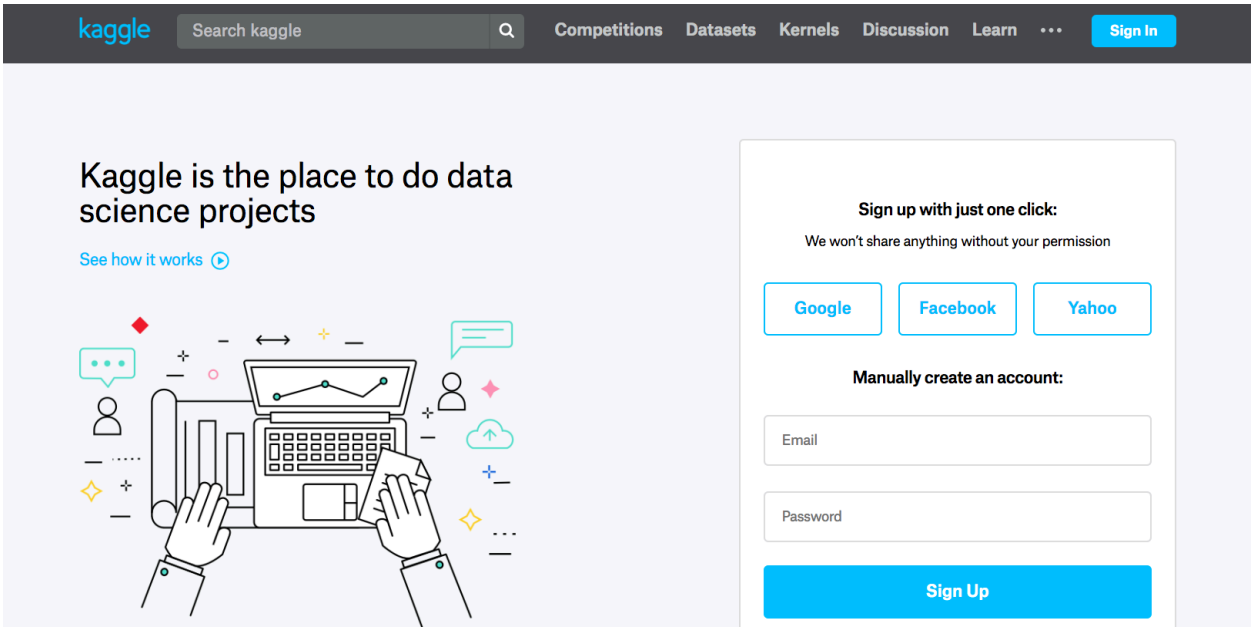


Fig. 6.10.1: Kaggle website

On the House Prices Prediction page, you can find the dataset (under the data tab), submit predictions, see your ranking, etc., The URL is right here:

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

[82] https://www.kaggle.com/c/house-prices-advanced-regression-techniques
[83] http://jse.amstat.org/v19n3/decock.pdf
[84] https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names
[85] https://www.kaggle.com

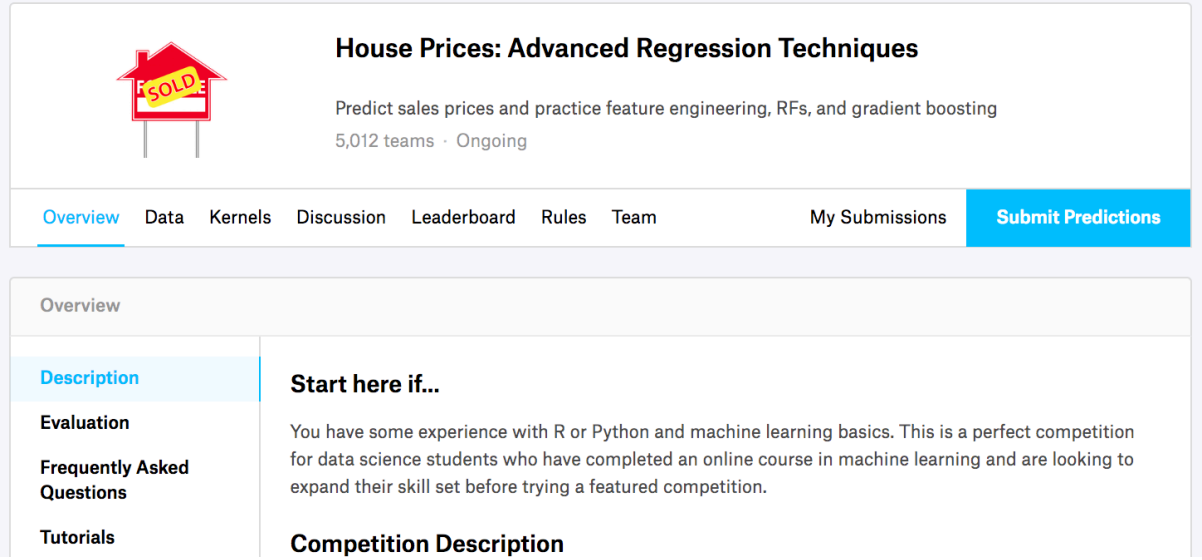


Fig. 6.10.2: House Price Prediction

6.10.2 Accessing and Reading Data Sets

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 represent multiple datatypes. Year of construction, for example, is represented with integers roof type is a discrete categorical feature, other features are represented with floating point numbers. And here is where reality comes in: for some examples, some data is 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). You can partition the training set to create a validation set, but you will only find out how you perform on the official test set when you upload your predictions and receive your score. The 'Data' tab on the competition tab has links to download the data.

We will read and process the data using `pandas`, an efficient data analysis toolkit[86], 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 d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
import pandas as pd
npx.set_np()
```

For convenience, we already downloaded the data and stored it in the `../data` directory. To load the two CSV (Comma Separated Values) files containing training and test data respectively we use Pandas.

```
train_data = pd.read_csv('../data/kaggle_house_pred_train.csv')
test_data = pd.read_csv('../data/kaggle_house_pred_test.csv')
```

The training dataset includes 1,460 examples, 80 features, and 1 label., the test data contains 1,459 examples and 80

[86] http://pandas.pydata.org/pandas-docs/stable/

features.

```
print(train_data.shape)
print(test_data.shape)
```

```
(1460, 81)
(1459, 80)
```

Let's take a look at the first 4 and last 2 features as well as the label (SalePrice) from the first 4 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 network.

```
all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

6.10.3 Data Preprocessing

As stated above, we have a wide variety of datatypes. Before we feed it into a deep network, we need to perform some amount of processing. Let us start with the numerical features. We begin by replacing missing values with the mean. This is a reasonable strategy if features are missing at random. To adjust them to a common scale, we rescale them to zero mean and unit variance. This is accomplished as follows:

$$x \leftarrow \frac{x - \mu}{\sigma} \tag{6.10.1}$$

To check that this transforms x to data with zero mean and unit variance simply calculate $\mathbf{E}[(x-\mu)/\sigma] = (\mu-\mu)/\sigma = 0$. To check the variance we use $\mathbf{E}[(x-\mu)^2] = \sigma^2$ and thus the transformed variable has unit variance. The reason for 'normalizing' the data is that it brings all features to the same order of magnitude. After all, we do not know *a priori* which features are likely to be relevant.

```
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 variables such as 'MSZoning'. We replace them by a one-hot encoding in the same manner as how we transformed multiclass classification data into a vector of 0 and 1. For instance, 'MSZoning' assumes the values 'RL' and 'RM'. They map into vectors $(1, 0)$ and $(0, 1)$ respectively. Pandas does this automatically for us.

```
# Dummy_na=True refers to a missing value being a legal eigenvalue, and
# creates an indicative 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 dataframe and convert it into MXNet's native `ndarray` representation for training.

```
n_train = train_data.shape[0]
train_features = np.array(all_features[:n_train].values)
test_features = np.array(all_features[n_train:].values)
train_labels = np.array(train_data.SalePrice.values).reshape(-1, 1)
```

6.10.4 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. More concretely, 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 (their, 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 metric used by the compeitition to measure the quality of submissions. After all, a small value δ of $\log y - \log \hat{y}$ translates into $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$. This leads to the following loss function:

$$L = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\log y_i - \log \hat{y}_i\right)^2} \tag{6.10.2}$$

```
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 here will rely on the Adam optimizer (a slight variant on SGD that we will describe in greater detail later). The main appeal of Adam vs vanilla SGD is that the Adam optimizer, 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. This will be covered in further detail later on when we discuss the details in Section 12.

```
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
```

6.10.5 k-Fold Cross-Validation

If you are reading in a linear fashion, you might recall that we intorduced k-fold cross-validation in the section where we discussed how to deal with model section (Section 6.4). 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 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
    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), axis=0)
            y_train = np.concatenate((y_train, y_part), axis=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):
```

(continues on next page)

(continued from previous page)

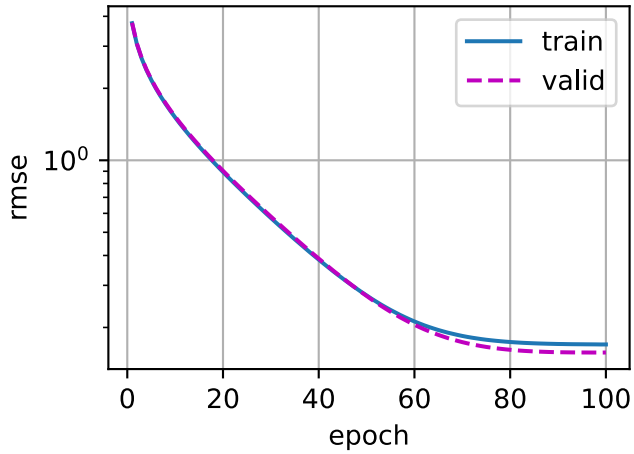
```
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
        data = get_k_fold_data(k, i, X_train, y_train)
        net = get_net()
        train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
                                   weight_decay, batch_size)
        train_l_sum += train_ls[-1]
        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',
                     legend=['train', 'valid'], yscale='log')
        print('fold %d, train rmse: %f, valid rmse: %f' % (
            i, train_ls[-1], valid_ls[-1]))
    return train_l_sum / k, valid_l_sum / k
```

6.10.6 Model Selection

In this example, we pick an un-tuned set of hyperparameters and leave it up to the reader to improve the model. Finding a good choice can take quite some time, depending on how many things one wants to optimize over. Within reason, the k-fold cross-validation approach is resilient against multiple testing. However, if we were to try out an unreasonably large number of options it might fail since we might just get lucky on the validation split with a particular set of hyperparameters.

```
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('%d-fold validation: avg train rmse: %f, avg valid rmse: %f'
      % (k, train_l, valid_l))
```

```
fold 0, train rmse: 0.169821, valid rmse: 0.157146
fold 1, train rmse: 0.162019, valid rmse: 0.188183
fold 2, train rmse: 0.164074, valid rmse: 0.168179
fold 3, train rmse: 0.167739, valid rmse: 0.154893
fold 4, train rmse: 0.162565, valid rmse: 0.182744
5-fold validation: avg train rmse: 0.165243, avg valid rmse: 0.170229
```



You will notice that sometimes the number of training errors for a set of hyper-parameters can be very low, while the number of errors for the K-fold cross-validation may be higher. This is an indicator that we are overfitting. Therefore, when we reduce the amount of training errors, we need to check whether the amount of errors in the k-fold cross-validation have also been reduced accordingly.

6.10.7 Predict and Submit

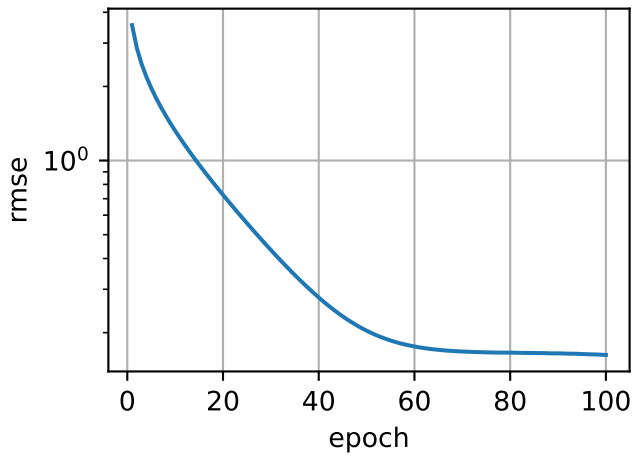
Now that we know what a good choice of hyperparameters should be, we might as well use all the data to train on it (rather than just $1 - 1/k$ of the data that is used in the cross-validation slices). The model that we obtain in this way can then be applied to the test set. Saving the estimates in a 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='rmse', yscale='log')
    print('train rmse %f' % train_ls[-1])
    # Apply the network to the test set
    preds = net(test_features).asnumpy()
    # Reformat it for 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)
```

Let us invoke our model. 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` (CSV is one of the file formats accepted by Kaggle):

```
train_and_pred(train_features, test_features, train_labels, test_data,
               num_epochs, lr, weight_decay, batch_size)
```

```
train rmse 0.162320
```



Next, we can submit our predictions on Kaggle and see how they compare to 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.

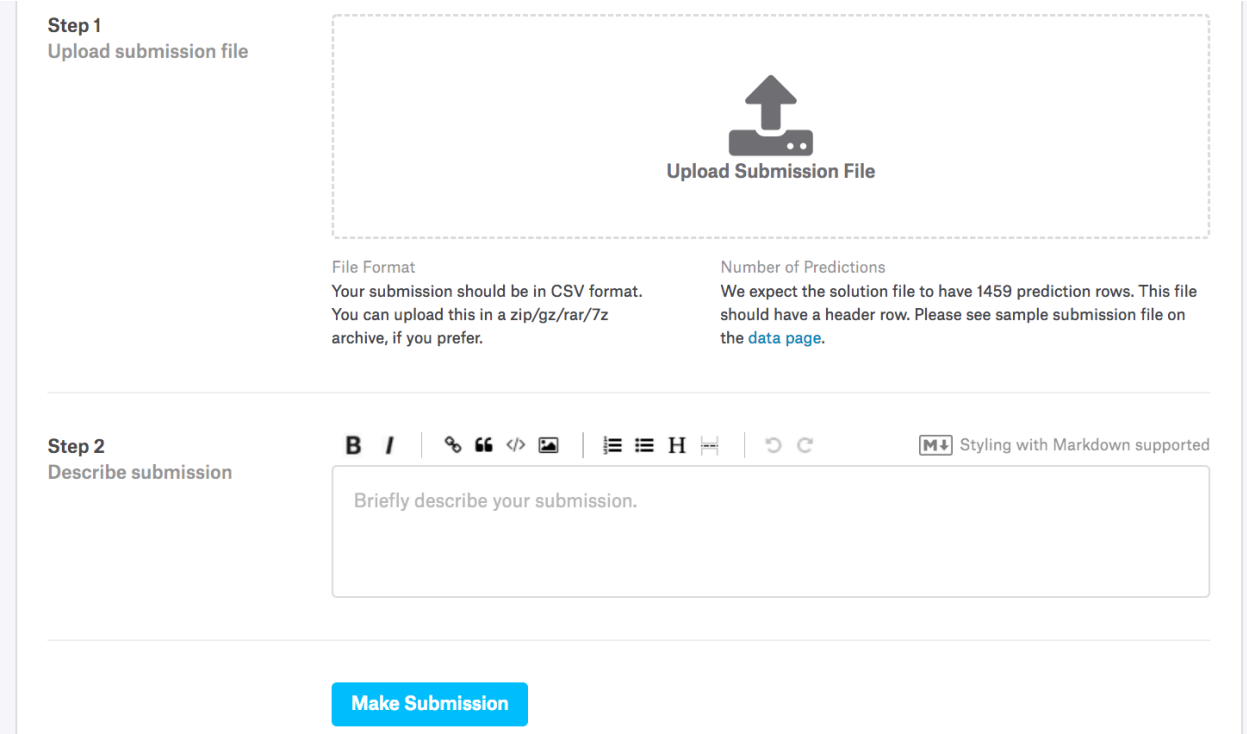


Fig. 6.10.3: Submitting data to Kaggle

6.10.8 Summary

- Real data often contains a mix of different datatypes and needs 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 variables into indicator variables allows us to treat them like vectors.
- We can use k-fold cross validation to select the model and adjust the hyper-parameters.
- Logarithms are useful for relative loss.

6.10.9 Exercises

1. Submit your predictions for this tutorial to Kaggle. How good are your predictions?
2. Can you improve your model by minimizing the log-price directly? What happens if you try to predict the log 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. Find a better representation to deal with missing values. Hint - What happens if you add an indicator variable?
5. Improve the score on Kaggle by tuning the hyperparameters through k-fold cross-validation.
6. Improve the score by improving the model (layers, regularization, dropout).
7. What happens if we do not standardize the continuous numerical features like we have done in this section?

6.10.10 Scan the QR Code to Discuss[87]

[87] https://discuss.mxnet.io/t/2346

CHAPTER

SEVEN

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 artifical 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 a multilayer perceptron from scratch and even showed how to leverage MXNet's Gluon library 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 combine the 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.

7.1 Layers and Blocks

When we first started talking about neural networks, we introduced linear models with a single output. Here, the entire model consists of just a single neuron. By itself, a single neuron takes some set of inputs, generates a corresponding (*scalar*) output, and 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, we showed how we could use linear algebra to efficiently express an entire *layer* of neurons. Layers too expect some inputs, generate corresponding outputs, and are described by a set of tunable parameters.

When we worked through softmax regression, a single *layer* was itself *the model*. However, when we subsequently introduced multilayer perceptrons, we developed models consisting of multiple layers. One interesting property of multilayer neural networks is that the *entire model* and its *constituent layers* share the same basic structure. The model takes the true inputs (as stated in the problem formulation), outputs predictions of the true outputs, and possesses parameters (the combined set of all parameters from all layers) Likewise any individual constituent layer in a multilayer perceptron ingests inputs (supplied by the previous layer) generates outputs (which form the inputs to the subsequent layer), and possesses a set of tunable parameters tht are updated with respect to the ultimate objective (using the signal that flows backwards through 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 will often want to express our model in terms of a components that are large than an indivudal layer. For example, when designing models, like ResNet-152, which possess hundreds (152, thus the name) of layers, implementing the network one layer at a time can grow tedious. Moreover, this concern is not just hypothetical—such deep networks

dominate numerous application areas, especally when training data is abundant. For example the ResNet architecture mentioned above (He et al.[88]) won the 2015 ImageNet and COCO computer vision compeititions for both recognition and detection. Deep networks with many layers arranged into components with various repeating patterns are now ubiquitous in other domains including natural language processing and speech.

To facilitate the implementation of networks consisting of components of arbitrary complecity, we introduce a new flexible concept: a neural network *block*. A block could describe a single neuron, a high-dimensional layer, or an arbitrarily-complex component consisting of multiple layers. From a software development, a `Block` is a class. Any subclass of `Block` must define a method called `forward` 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 `backward` method, for purposes of calculating gradients. Fortunately, due to some behind-the-scenes magic supplied by the autograd `autograd` package (introduced in Section 4) when defining our own `Block` typically requires only that we worry about parameters and the `forward` function.

One benefit of working with the `Block` abstraction is that they can be combined into larger artifacts, often recursively, e.g., as illustrated in the following diagram:

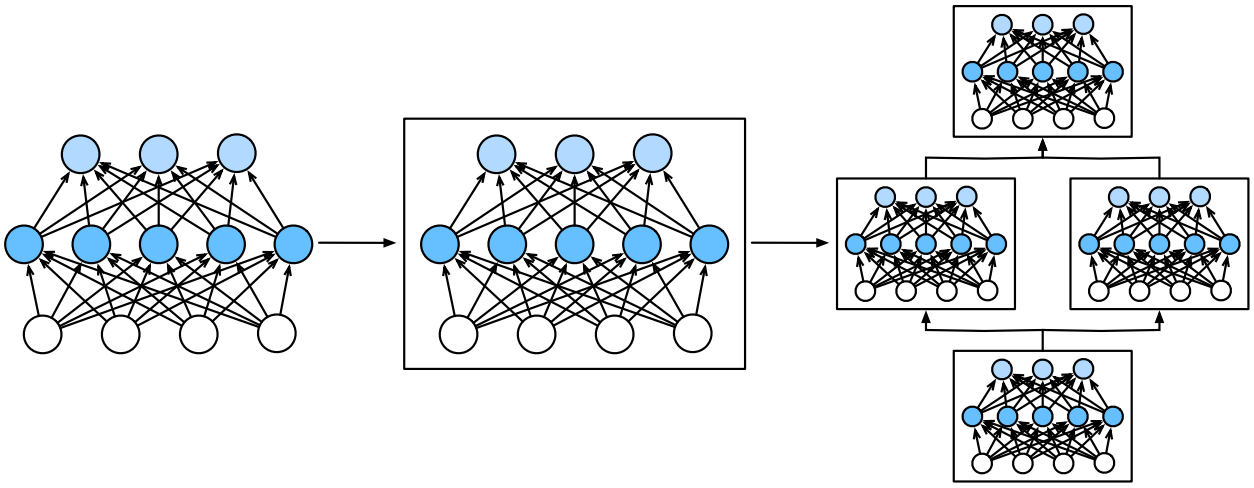


Fig. 7.1.1: Multiple layers are combined into blocks

By defining code to generate Blocks of arbitrary complexity on demand, we can write surprisingly compact code and still implement complex neural networks.

To begin, we revisit the Blocks that played a role in our implementation of the multilayer perceptron (Section 6.3). The following code generates a network with one fully-connected hidden layer containing 256 units followed by a ReLU activation, and then another fully-connected layer consisting of 10 units (with no activation function). Because there are no more layers, this last 10-unit layer is regarded as the *output layer* and its outputs are also the model's output.

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

x = np.random.uniform(size=(2, 20))

net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(x)
```

[88] https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf

```
array([[ 0.06240272, -0.03268593,  0.02582653,  0.02254182, -0.03728798,
        -0.04253786,  0.00540613, -0.01364186, -0.09915452, -0.02272738],
       [ 0.02816677, -0.03341204,  0.03565666,  0.02506382, -0.04136416,
        -0.04941845,  0.01738528,  0.01081961, -0.09932579, -0.01176298]])
```

In this example, as in previous chapters, our model consists of an object returned by the `nn.Sequential` constructor. After instantiating a `nn.Sequential` and storing the `net` variable, we repeatedly called its `add` method, appending layers in the order that they should be executed. We suspect that you might have already understood *more or less* what was going on here the first time you saw this code. You may even have understood it well enough to modify the code and design your own networks. However, the details regarding what exactly happens inside `nn.Sequential` have remained mysterious so far.

In short, `nn.Sequential` just defines a special kind of Block. Specifically, an `nn.Sequential` maintains a list of constitutent `Blocks`, stored in a particular order. You might think of `nnSequential` as your first meta-Block. The `add` method simply facilitates the addition of each successive `Block` to the list. Note that each our layers are instances of the `Dense` class which is itself a subclass of `Block`. The `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.

Before we dive in to implementing our own custom `Block`, we briefly summarize the basic functionality that each `Block` must perform the following duties:

1. Ingest input data as arguments to its `forward` function.
2. Generate an output via the value returned by its `forward` function. Note that the output may have a different shape from the input. For example, the first Dense 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 `backward` method. Typically this happens automatically.
4. Store and provide access to those parameters necessary to execute the `forward` computation.
5. Initialize these parameters as needed.

7.1.1 A Custom Block

Perhaps the easiest way to develop intuition about how `nn.Block` works is to just dive right in and implement one ourselves. In the following snippet, instead of relying on `nn.Sequential`, we just code up a Block from scratch that implements a multilayer perceptron with one hidden layer, 256 hidden nodes, and 10 outputs.

Our `MLP` class below inherits the `Block` class. While we rely on some predefined methods in the parent class, we need to supply our own `__init__` and `forward` functions to uniquely define the behavior of our model.

```
from mxnet.gluon import nn

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 parameters can
        # also be specified when constructing an instance, such as the model
        # parameter, params, described in the following sections
        super(MLP, self).__init__(**kwargs)
```

(continues on next page)

(continued from previous page)

```
        self.hidden = nn.Dense(256, activation='relu')  # Hidden layer
        self.output = nn.Dense(10)  # Output layer

    # Define the forward computation of the model, that is, how to return the
    # required model output based on the input x
    def forward(self, x):
        return self.output(self.hidden(x))
```

This code may be easiest to understand by working backwards from `forward`. Note that the `forward` method takes as input `x`. The forward method first evaluates `self.hidden(x)` to produce the hidden representation, passing this output as the input to the output layer `self.output( ... )`.

The constituent layers of each `MLP` must be instance-level variables. After all, if we instantiated two such models `net1` and `net2` and trained them on different data, we would expect them to them to represent two different learned models.

The `__init__` method is the most natural place to instantiate the layers that we subsequently invoke on each call to the `forward` method. Note that before getting on with the interesting parts, our customized `__init__` method must invoke the parent class's init method: `super(MLP, self).__init__(**kwargs)` to save us from reimplementing boilerplate code applicable to most Blocks. Then, all that is left is to instantiate our two `Dense` layers, assigning them to `self.hidden` and `self.output`, respectively. Again note that when dealing with standard functionality like this, we do not have to worry about backpropagation, since the `backward` method is generated for us automatically. The same goes for the `initialize` method. Let us try this out:

```
net = MLP()
net.initialize()
net(x)
```

```
array([[-0.03989594, -0.1041471 ,  0.06799038,  0.05245074,  0.02526059,
        -0.00640342,  0.04182098, -0.01665319, -0.02067346, -0.07863817],
       [-0.03612847, -0.07210436,  0.09159479,  0.07890771,  0.02494172,
        -0.01028665,  0.01732428, -0.02843242,  0.03772651, -0.06671704]])
```

As we argued earlier, the primary virtue of the `Block` abstraction is its versatility. We can subclass `Block` to create layers (such as the `Dense` class provided by Gluon), entire models (such as the `MLP` class implemented above), or various components of intermediate complexity, a pattern that we will lean on heavily throughout the next chapters on convolutinoal neural networks.

7.1.2 The Sequential Block

As we described earlier, the `Sequential` class itself is also just a subclass of `Block`, designed specifically for daisy-chaining other Blocks together. All we need to do to implement our own `MySequential` block is to define a few convenience functions: 1. An `add` method for appending Blocks one by one to a list. 2. A `forward` method to pass inputs through the chain of Blocks (in the order of addition).

The following `MySequential` class delivers the same functionality as Gluon's default Sequential class:

```
class MySequential(nn.Block):
    def __init__(self, **kwargs):
        super(MySequential, self).__init__(**kwargs)

    def add(self, block):
        # Here, block is an instance of a Block subclass, and we assume it has
        # a unique name. We save it in the member variable _children of the
```

(continues on next page)

(continued from previous page)

```
        # 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
```

At its core is the `add` method. It adds any block to the ordered dictionary of children. These are then executed in sequence when forward propagation is invoked. Let us see what the MLP looks like now.

```
net = MySequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(x)
```

```
array([[-0.07645682, -0.01130233,  0.04952145, -0.04651389, -0.04131573,
        -0.05884133, -0.0621381 ,  0.01311472, -0.01379425, -0.02514282],
       [-0.05124625,  0.00711231, -0.00155935, -0.07555379, -0.06675334,
        -0.01762914,  0.00589084,  0.01447191, -0.04330775,  0.03317726]])
```

Indeed, it can be observed that the use of the `MySequential` class is no different from the use of the Sequential class described in Section 6.3.

7.1.3 Blocks with Code

Although the Sequential class can make model construction easier, and you do not need to define the `forward` method, directly inheriting the Block class can greatly expand the flexibility of model construction. In particular, we will use Python's control flow within the forward method. While we are at it, we need to introduce another concept, that of the *constant* parameter. These are parameters that are not used when invoking backprop. This sounds very abstract but here's what is really going on. Assume that we have some function

$$f(\mathbf{x}, \mathbf{w}) = 3 \cdot \mathbf{w}^\top \mathbf{x}. \tag{7.1.1}$$

In this case 3 is a constant parameter. We could change 3 to something else, say c via

$$f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}. \tag{7.1.2}$$

Nothing has really changed, except that we can adjust the value of c. It is still a constant as far as $\mathbf{w}$ and $\mathbf{x}$ are concerned. However, since Gluon does not know about this beforehand, it is worth while to give it a hand (this makes the code go faster, too, since we are not sending the Gluon engine on a wild goose chase after a parameter that does not change). `get_constant` is the method that can be used to accomplish this. Let us see what this looks like in practice.

```
class FancyMLP(nn.Block):
    def __init__(self, **kwargs):
        super(FancyMLP, self).__init__(**kwargs)
        # Random weight parameters created with the get_constant are not
```

(continues on next page)

(continued from previous page)

```
        # iterated 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 constant parameters created, 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)
        # Here in Control flow, we need to call asscalar to return the scalar
        # for comparison
        while np.abs(x).sum() > 1:
            x /= 2
        if np.abs(x).sum() < 0.8:
            x *= 10
        return x.sum()
```

In this `FancyMLP` model, we used constant weight `Rand_weight` (note that it is not a model parameter), performed a matrix multiplication operation (`np.dot<`), and reused the *same* `Dense` layer. Note that this is very different from using two dense layers with different sets of parameters. Instead, we used the same network twice. Quite often in deep networks one also says that the parameters are *tied* when one wants to express that multiple parts of a network share the same parameters. Let us see what happens if we construct it and feed data through it.

```
net = FancyMLP()
net.initialize()
net(x)
```

```
array(5.2637568)
```

There is no reason why we couldn't mix and match these ways of build a network. Obviously the example below resembles more a chimera, or less charitably, a Rube Goldberg Machine[89]. That said, it combines examples for building a block from individual blocks, which in turn, may be blocks themselves. Furthermore, we can even combine multiple strategies inside the same forward function. To demonstrate this, here's the network.

```
class NestMLP(nn.Block):
    def __init__(self, **kwargs):
        super(NestMLP, self).__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), FancyMLP())

chimera.initialize()
chimera(x)
```

[89] https://en.wikipedia.org/wiki/Rube_Goldberg_machine

```
array(0.97720534)
```

7.1.4 Compilation

The avid reader is probably starting to worry about the efficiency of this. After all, we have lots of dictionary lookups, code execution, and lots of other Pythonic things going on in what is supposed to be a high performance deep learning library. The problems of Python's Global Interpreter Lock[90] are well known. In the context of deep learning it means that we have a super fast GPU (or multiple of them) which might have to wait until a puny single CPU core running Python gets a chance to tell it what to do next. This is clearly awful and there are many ways around it. The best way to speed up Python is by avoiding it altogether.

Gluon does this by allowing for Hybridization (Section 13.1). In it, the Python interpreter executes the block the first time it is invoked. The Gluon runtime records what is happening and the next time around it short circuits any calls to Python. This can accelerate things considerably in some cases but care needs to be taken with control flow. We suggest that the interested reader skip forward to the section covering hybridization and compilation after finishing the current chapter.

7.1.5 Summary

- Layers are blocks
- Many layers can be a block
- Many blocks can be a block
- Code can be a block
- Blocks take are of a lot of housekeeping, such as parameter initialization, backprop and related issues.
- Sequential concatenations of layers and blocks are handled by the eponymous `Sequential` block.

7.1.6 Exercises

1. What kind of error message will you get when calling an `__init__` method whose parent class not in the `__init__` function of the parent class?
2. What kinds of problems will occur if you remove the `asscalar` function in the `FancyMLP` class?
3. What kinds of problems will occur if you change `self.net` defined by the Sequential instance in the `NestMLP` class to `self.net = [nn.Dense(64, activation='relu'), nn. Dense(32, activation='relu')]`?
4. 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 pass (this is also called a parallel block).
5. 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.

7.1.7 Scan the QR Code to Discuss[91]

[90] https://wiki.python.org/moin/GlobalInterpreterLock
[91] https://discuss.mxnet.io/t/2325

7.2 Parameter Management

The ultimate goal of training deep networks is to find good parameter values for a given architecture. When everything is standard, the `nn.Sequential` class is a perfectly good tool for it. However, very few models are entirely standard and most scientists want to build things that are novel. This section shows how to manipulate parameters. In particular we will cover the following aspects:

- Accessing parameters for debugging, diagnostics, to visualize them or to save them is the first step to understanding how to work with custom models.
- Secondly, we want to set them in specific ways, e.g., for initialization purposes. We discuss the structure of parameter initializers.
- Lastly, we show how this knowledge can be put to good use by building networks that share some parameters.

As always, we start from our trusty Multilayer Perceptron with a hidden layer. This will serve as our choice for demonstrating the various features.

```
from mxnet import init, 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()  # Use the default initialization method

x = np.random.uniform(size=(2, 20))
net(x)  # Forward computation
```

```
array([[ 0.06240272, -0.03268593,  0.02582653,  0.02254182, -0.03728798,
        -0.04253786,  0.00540613, -0.01364186, -0.09915452, -0.02272738],
       [ 0.02816677, -0.03341204,  0.03565666,  0.02506382, -0.04136416,
        -0.04941845,  0.01738528,  0.01081961, -0.09932579, -0.01176298]])
```

7.2.1 Parameter Access

In the case of a Sequential class we can access the parameters with ease, simply by indexing each of the layers in the network. The params variable then contains the required data. Let us try this out in practice by inspecting the parameters of the first layer.

```
print(net[0].params)
print(net[1].params)
```

```
dense0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
dense1_ (
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

The output tells us a number of things. Firstly, the layer consists of two sets of parameters: `dense0_weight` and `dense0_bias`, as we would expect. They are both single precision and they have the necessary shapes that we would expect from the first layer, given that the input dimension is 20 and the output dimension 256. In particular the names of the parameters are very useful since they allow us to identify parameters *uniquely* even in a network of hundreds of layers and with nontrivial structure. The second layer is structured accordingly.

Targeted Parameters

In order to do something useful with the parameters we need to access them, though. There are several ways to do this, ranging from simple to general. Let us look at some of them.

```
print(net[1].bias)
print(net[1].bias.data())
```

```
Parameter dense1_bias (shape=(10,), dtype=float32)
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

The first returns the bias of the second layer. Since this is an object containing data, gradients, and additional information, we need to request the data explicitly. Note that the bias is all 0 since we initialized the bias to contain all zeros. Note that we can also access the parameters by name, such as `dense0_weight`. This is possible since each layer comes with its own parameter dictionary that can be accessed directly. Both methods are entirely equivalent but the first method leads to much more readable code.

```
print(net[0].params['dense0_weight'])
print(net[0].params['dense0_weight'].data())
```

```
Parameter dense0_weight (shape=(256, 20), dtype=float32)
[[ 0.06700657 -0.00369488  0.0418822  ... -0.05517294 -0.01194733
  -0.00369594]
 [-0.03296221 -0.04391347  0.03839272 ...  0.05636378  0.02545484
  -0.007007  ]
 [-0.0196689   0.01582889 -0.00881553 ...  0.01509629 -0.01908049
  -0.02449339]
 ...
 [-0.02055008 -0.02618652  0.06762936 ... -0.02315108 -0.06794678
  -0.04618235]
 [ 0.02802853  0.06672969  0.05018687 ... -0.02206502 -0.01315478
  -0.03791244]
 [-0.00638592  0.00914261  0.06667828 ... -0.00800052  0.03406764
  -0.03954004]]
```

Note that the weights are nonzero. This is by design since they were randomly initialized when we constructed the network. `data` is not the only function that we can invoke. For instance, we can compute the gradient with respect to

the parameters. It has the same shape as the weight. However, since we did not invoke backpropagation yet, the values are all 0.

```
net[0].weight.grad()
```

```
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.]])
```

All Parameters at Once

Accessing parameters as described above can be a bit tedious, in particular if we have more complex blocks, or blocks of blocks (or even blocks of blocks of blocks), since we need to walk through the entire tree in reverse order to how the blocks were constructed. To avoid this, blocks come with a method `collect_params` which grabs all parameters of a network in one dictionary such that we can traverse it with ease. It does so by iterating over all constituents of a block and calls `collect_params` on subblocks as needed. To see the difference consider the following:

```
# parameters only for the first layer
print(net[0].collect_params())
# parameters of the entire network
print(net.collect_params())
```

```
dense0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
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)
)
```

This provides us with a third way of accessing the parameters of the network. If we wanted to get the value of the bias term of the second layer we could simply use this:

```
net.collect_params()['dense1_bias'].data()
```

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

Throughout the book we will see how various blocks name their subblocks (Sequential simply numbers them). This makes it very convenient to use regular expressions to filter out the required parameters.

```
print(net.collect_params('.*weight'))
print(net.collect_params('dense0.*'))
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
)
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
```

Rube Goldberg strikes again

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 we 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 i in range(4):
        net.add(block1())
    return net

rgnet = nn.Sequential()
rgnet.add(block2())
rgnet.add(nn.Dense(10))
rgnet.initialize()
rgnet(x)
```

```
array([[-4.1923025e-09,  1.9830502e-09,  8.9444063e-10,  6.2912990e-09,
        -3.3241778e-09,  5.4330038e-09,  1.6013515e-09, -3.7408681e-09,
         8.5468477e-09, -6.4805539e-09],
       [-3.7507064e-09,  1.4866974e-09,  6.8314709e-10,  5.6925784e-09,
        -2.6349172e-09,  4.8626667e-09,  1.4280275e-09, -3.4603027e-09,
         7.4127922e-09, -5.7896132e-09]])
```

Now that we are done designing the network, let us see how it is organized. `collect_params` provides us with this information, both in terms of naming and in terms of logical structure.

```
print(rgnet.collect_params)
print(rgnet.collect_params())
```

```
<bound method Block.collect_params of Sequential(
  (0): Sequential(
    (0): Sequential(
      (0): Dense(20 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (1): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
```

(continues on next page)

(continued from previous page)

```
      (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))
    )
  )
  (1): Dense(16 -> 10, linear)
)>
sequential1_ (
  Parameter dense2_weight (shape=(32, 20), 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 generated, we can also access them accordingly. For instance, to access the first major block, within it the second subblock and then within it, in turn the bias of the first layer, we perform the following.

```
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.])
```

7.2.2 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 initialization in Section 6.8. By default, MXNet initializes the weight matrices uniformly by drawing from $U[-0.07, 0.07]$ and the bias parameters are all set to 0. However, we often need to use other methods to initialize the weights. MXNet's `init` module provides a variety of preset initialization methods, but if we want something out of the ordinary, we need a bit of extra work.

Built-in Initialization

Let us begin with the built-in initializers. The code below initializes all parameters with Gaussian random variables.

```
# force_reinit ensures that the variables are initialized again, regardless of
# whether they were already initialized previously
net.initialize(init=init.Normal(sigma=0.01), force_reinit=True)
net[0].weight.data()[0]
```

```
array([ 0.02212206,  0.00774004,  0.0104344 ,  0.01183925,  0.01891711,
       -0.01234741, -0.01771029, -0.00451384,  0.00579384, -0.01856082,
       -0.01976879, -0.00208019,  0.00244422, -0.00037161, -0.0048775 ,
       -0.00022617,  0.00574614,  0.01466126,  0.0068629 ,  0.00354961])
```

If we wanted to initialize all parameters to 1, we could do this simply by changing the initializer to `Constant(1)`.

```
net.initialize(init=init.Constant(1), force_reinit=True)
net[0].weight.data()[0]
```

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

If we want to initialize only a specific parameter in a different manner, we can simply set the initializer only for the appropriate subblock (or parameter) for that matter. For instance, below we initialize the second layer to a constant value of 42 and we use the `Xavier` initializer for the weights of the first layer.

```
net[1].initialize(init=init.Constant(42), force_reinit=True)
net[0].weight.initialize(init=init.Xavier(), force_reinit=True)
print(net[1].weight.data()[0,0])
print(net[0].weight.data()[0])
```

```
42.0
[ 0.05148788 -0.09426671 -0.08199714 -0.07816338  0.03051682 -0.0699063
  0.04725215 -0.02083874  0.00679845  0.07578462  0.07648772  0.0107249
 -0.14014682  0.10345995 -0.09108455 -0.14639433  0.1375255   0.12978908
 -0.00066644 -0.09715894]
```

Custom Initialization

Sometimes, the initialization methods we need are not provided in the `init` module. At this point, we can implement a subclass of the `Initializer` class so that we can use it like any other initialization method. Usually, we only need to implement the `_init_weight` function and modify the incoming `ndarray` according to the initial result. In the example below, we pick a decidedly bizarre and nontrivial distribution, just to prove the point. We draw the coefficients from the following distribution:

$$w \sim \begin{cases} U[5, 10] & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ U[-10, -5] & \text{with probability } \frac{1}{4} \end{cases} \tag{7.2.1}$$

```
class MyInit(init.Initializer):
    def _init_weight(self, name, data):
        print('Init', name, data.shape)
        data[:] = np.random.uniform(-10, 10, data.shape)
        data *= np.abs(data) >= 5

net.initialize(MyInit(), force_reinit=True)
net[0].weight.data()[0]
```

```
Init dense0_weight (256, 20)
Init dense1_weight (10, 256)
```

```
array([-0.       ,  0.       ,  5.929554 , -7.169097 ,  0.       ,
       -6.583863 ,  0.       ,  0.       , -0.       , -7.85941  ,
       -0.       ,  5.702038 , -7.1187963, -0.       ,  9.750004 ,
       -0.       ,  0.       , -8.1825485, -0.       , -0.       ])
```

If even this functionality is insufficient, we can set parameters directly. Since `data()` returns an `ndarray` we can access it just like any other matrix. 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.

```
net[0].weight.data()[:] += 1
net[0].weight.data()[0,0] = 42
net[0].weight.data()[0]
```

```
array([42.       ,  1.       ,  6.929554 , -6.169097 ,  1.       ,
       -5.583863 ,  1.       ,  1.       ,  1.       , -6.85941  ,
        1.       ,  6.702038 , -6.1187963,  1.       , 10.750004 ,
        1.       ,  1.       , -7.1825485,  1.       ,  1.       ])
```

7.2.3 Tied Parameters

In some cases, we want to share model parameters across multiple layers. For instance when we want to find good word embeddings we may decide to use the same parameters both for encoding and decoding of words. We discussed one such case when we introduced Section 7.1. Let us see how to do this a bit more 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 such that we can reference 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
```

(continues on next page)

(continued from previous page)

```
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])
```

```
[1. 1. 1. 1. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1.]
```

The above example shows that the parameters of the second and third layer are tied. They are identical rather than just being equal. That is, by changing one of the parameters the other one changes, too. What happens to the gradients is quite ingenious. Since the model parameters contain gradients, the gradients of the second hidden layer and the third hidden layer are accumulated in the `shared.params.grad( )` during backpropagation.

7.2.4 Summary

- We have several ways to access, initialize, and tie model parameters.
- We can use custom initialization.
- Gluon has a sophisticated mechanism for accessing parameters in a unique and hierarchical manner.

7.2.5 Exercises

1. Use the FancyMLP defined in Section 7.1 and access the parameters of the various layers.
2. Look at the MXNet documentation[92] and explore different initializers.
3. Try accessing the model parameters after `net.initialize()` and before `net(x)` to observe the shape of the model parameters. What changes? Why?
4. Construct a multilayer perceptron containing a shared parameter layer and train it. During the training process, observe the model parameters and gradients of each layer.
5. Why is sharing parameters a good idea?

7.2.6 Scan the QR Code to Discuss[93]

7.3 Deferred Initialization

In the previous examples we played fast and loose with setting up our networks. In particular we did the following things that *shouldn't* work:

[92] http://beta.mxnet.io/api/gluon-related/mxnet.initializer.html
[93] https://discuss.mxnet.io/t/2326

- We defined the network architecture with no regard to the input dimensionality.
- We added layers without regard to the output dimension of the previous layer.
- We even 'initialized' these parameters without knowing how many parameters were to initialize.

All of those things sound impossible and indeed, they are. After all, there is no way MXNet (or any other framework for that matter) could predict what the input dimensionality of a network would be. Later on, when working with convolutional networks and images this problem will become even more pertinent, since the input dimensionality (i.e., the resolution of an image) will affect the dimensionality of subsequent layers at a long range. 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 statistical modeling. In what follows, we will discuss how this works using initialization as an example. After all, we cannot initialize variables that we do not know exist.

7.3.1 Instantiating a Network

Let us see what happens when we instantiate a network. We start with our trusty MLP as before.

```
from mxnet import init, np, npx
from mxnet.gluon import nn
npx.set_np()

def getnet():
    net = nn.Sequential()
    net.add(nn.Dense(256, activation='relu'))
    net.add(nn.Dense(10))
    return net

net = getnet()
```

At this point the network does not really know yet what the dimensionalities of the various parameters should be. All one could tell at this point is that each layer needs weights and bias, albeit of unspecified dimensionality. If we try accessing the parameters, that is exactly what happens.

```
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)
)
```

In particular, trying to access `net[0].weight.data()` at this point would trigger a runtime error stating that the network needs initializing before it can do anything. Let us see whether anything changes after we initialize the parameters:

```
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 really changed. Only once we provide the network with some data do we see a difference. Let us try it out.

```
x = np.random.uniform(size=(2, 20))
net(x)  # Forward computation

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)
)
```

The main difference to before is that as soon as we knew the input dimensionality, $\mathbf{x} \in \mathbb{R}^{20}$ it was possible to define the weight matrix for the first layer, i.e., $\mathbf{W}_1 \in \mathbb{R}^{256 \times 20}$. With that out of the way, we can progress to the second layer, define its dimensionality to be 10×256 and so on through the computational graph and bind all the dimensions as they become available. Once this is known, we can proceed by initializing parameters. This is the solution to the three problems outlined above.

7.3.2 Deferred Initialization in Practice

Now that we know how it works in theory, let us see when the initialization is actually triggered. In order to do so, we mock up an initializer which does nothing but report a debug message stating when it was invoked and with which parameters.

```
class MyInit(init.Initializer):
    def _init_weight(self, name, data):
        print('Init', name, data.shape)
        # The actual initialization logic is omitted here

net = getnet()
net.initialize(init=MyInit())
```

Note that, although `MyInit` will print information about the model parameters when it is called, the above `initialize` function does not print any information after it has been executed. Therefore there is no real initialization parameter when calling the `initialize` function. Next, we define the input and perform a forward calculation.

```
x = np.random.uniform(size=(2, 20))
y = net(x)
```

```
Init dense2_weight (256, 20)
Init dense3_weight (10, 256)
```

At this time, information on the model parameters is printed. When performing a forward calculation based on the input `x`, the system can automatically infer the shape of the weight parameters of all layers based on the shape of the input.

Once the system has created these parameters, it calls the `MyInit` instance to initialize them before proceeding to the forward calculation.

Of course, this initialization will only be called when completing the initial forward calculation. After that, we will not re-initialize when we run the forward calculation `net(x)`, so the output of the `MyInit` instance will not be generated again.

```
y = net(x)
```

As mentioned at the beginning of this section, deferred initialization can also cause confusion. Before the first forward calculation, we were unable to directly manipulate the model parameters, for example, we could not use the `data` and `set_data` functions to get and modify the parameters. Therefore, we often force initialization by sending a sample observation through the network.

7.3.3 Forced Initialization

Deferred initialization does not occur if the system knows the shape of all parameters when calling the `initialize` function. This can occur in two cases:

- We have already seen some data and we just want to reset the parameters.
- We specified all input and output dimensions of the network when defining it.

The first case works just fine, as illustrated below.

```
net.initialize(init=MyInit(), force_reinit=True)
```

```
Init dense2_weight (256, 20)
Init dense3_weight (10, 256)
```

The second case requires us to specify the remaining set of parameters when creating the layer. For instance, for dense layers we also need to specify the `in_units` so that initialization can occur immediately once `initialize` is called.

```
net = nn.Sequential()
net.add(nn.Dense(256, in_units=20, activation='relu'))
net.add(nn.Dense(10, in_units=256))

net.initialize(init=MyInit())
```

```
Init dense4_weight (256, 20)
Init dense5_weight (10, 256)
```

7.3.4 Summary

- Deferred initialization is a good thing. It allows Gluon to set many things automagically and it removes a great source of errors from defining novel network architectures.
- We can override this by specifying all implicitly defined variables.
- Initialization can be repeated (or forced) by setting the `force_reinit=True` flag.

7.3.5 Exercises

1. What happens if you specify only parts of the input dimensions. Do you still 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 parameter tying.

7.3.6 Scan the QR Code to Discuss[94]

7.4 Custom Layers

One of the reasons for the success of deep learning can be found in the wide range of layers that can be used in a deep network. This allows for a tremendous degree of customization and adaptation. For instance, scientists have invented layers for images, text, pooling, loops, dynamic programming, even for computer programs. Sooner or later you will encounter a layer that does not exist yet in Gluon, or even better, you will eventually invent a new layer that works well for your problem at hand. This is when it is time to build a custom layer. This section shows you how.

7.4.1 Layers without Parameters

Since this is slightly intricate, we start with a custom layer (aka Block) that does not have any inherent parameters. Our first step is very similar to when we introduced blocks in Section 7.1. The following `CenteredLayer` class constructs a layer that subtracts the mean from the input. We build it by inheriting from the Block class and implementing the `forward` method.

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

class CenteredLayer(nn.Block):
    def __init__(self, **kwargs):
        super(CenteredLayer, self).__init__(**kwargs)

    def forward(self, x):
        return x - x.mean()
```

To see how it works let us feed some data into the layer.

```
layer = CenteredLayer()
layer(np.array([1, 2, 3, 4, 5]))
```

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

[94] https://discuss.mxnet.io/t/2327

We can also use it to construct more complex models.

```
net = nn.Sequential()
net.add(nn.Dense(128), CenteredLayer())
net.initialize()
```

Let us see whether the centering layer did its job. For that we send random data through the network and check whether the mean vanishes. Note that since we are dealing with floating point numbers, we are going to see a very small albeit typically nonzero number.

```
y = net(np.random.uniform(size=(4, 8)))
y.mean()
```

```
array(3.783498e-10)
```

7.4.2 Layers with Parameters

Now that we know how to define layers in principle, let us define layers with parameters. These can be adjusted through training. In order to simplify things for an avid deep learning researcher the `Parameter` class and the `ParameterDict` dictionary provide some basic housekeeping functionality. In particular, they govern access, initialization, sharing, saving and loading model parameters. For instance, this way we do not need to write custom serialization routines for each new custom layer.

For instance, we can use the member variable `params` of the `ParameterDict` type that comes with the Block class. It is a dictionary that maps string type parameter names to model parameters in the `Parameter` type. We can create a `Parameter` instance from `ParameterDict` via the `get` function.

```
params = gluon.ParameterDict()
params.get('param2', shape=(2, 3))
params
```

```
(
  Parameter param2 (shape=(2, 3), dtype=<class 'numpy.float32'>)
)
```

Let us use this to implement our own version of the dense layer. It has two parameters - bias and weight. To make it a bit nonstandard, we bake in the ReLU activation as default. Next, we implement a fully connected layer with both weight and bias parameters. It uses ReLU as an activation function, where `in_units` and `units` are the number of inputs and the number of outputs, respectively.

```
class MyDense(nn.Block):
    # units: the number of outputs in this layer; in_units: the number of
    # inputs in this layer
    def __init__(self, units, in_units, **kwargs):
        super(MyDense, self).__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()) + self.bias.data()
        return npx.relu(linear)
```

Naming the parameters allows us to access them by name through dictionary lookup later. It is a good idea to give them instructive names. 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 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 dense layer. The only exception is that in our case size inference is not automagic. Please consult the MXNet documentation[95] for details on how to do this.

```
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 ]])
```

7.4.3 Summary

- We can design custom layers via the Block class. This is more powerful than defining a block factory, since it can be invoked in many contexts.
- Blocks can have local parameters.

7.4.4 Exercises

1. Design a layer that learns an affine transform of the data, i.e., it removes the mean and learns an additive parameter instead.
2. Design a layer that takes an input and computes a tensor reduction, i.e., it returns $y_k = \sum_{i,j} W_{ijk} x_i x_j$.
3. Design a layer that returns the leading half of the Fourier coefficients of the data. Hint - look up the `fft` function in MXNet.

[95] http://www.mxnet.io

7.4.5 Scan the QR Code to Discuss[96]

7.5 File I/O

So far we discussed how to process data, how to build, train and test deep learning models. However, at some point we are likely happy with what we obtained and we want to save the results for later use and distribution. Likewise, when running a long training process it is best practice to save intermediate results (checkpointing) to ensure that we do not lose several days worth of computation when tripping over the power cord of our server. At the same time, we might want to load a pretrained model (e.g., we might have word embeddings for English and use it for our fancy spam classifier). For all of these cases we need to load and store both individual weight vectors and entire models. This section addresses both issues.

7.5.1 Load and Save `ndarrays`

In its simplest form, we can directly use the `load` and `save` functions to store and read `ndarrays` separately. This works just as expected.

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

x = np.arange(4)
npx.save('x-file', x)
```

Then, we read the data from the stored file back into memory.

```
x2 = npx.load('x-file')
x2
```

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

We can also store a list of `ndarrays` 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 a string to an `ndarray`. This is convenient, for instance when we want to read or write all the weights in a model.

[96] https://discuss.mxnet.io/t/2328

```
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.])}
```

7.5.2 Gluon Model Parameters

Saving individual weight vectors (or other `ndarray` 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. Writing a script that collects all the terms and matches them to an architecture is quite some work. For this reason Gluon provides built-in functionality to load and save entire networks rather than just single weight vectors. An important detail to note is that this saves model *parameters* and not the entire model. I.e. 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 quite so easily (there is a way to do this for compiled models - please refer to the MXNet documentation[97] for the technical details on it). The result is that in order to reinstate a model we need to generate the architecture in code and then load the parameters from disk. The deferred initialization (Section 7.3) is quite advantageous here since we can simply define a model without the need to put actual values in place. Let us start with our favorite 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 check whether we are able to recover the model we instantiate a clone of the original MLP model. Unlike the random initialization of model parameters, here 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 computation result of the same input `x` should be the same. Let us verify this.

```
yclone = clone(x)
yclone == y
```

[97] http://www.mxnet.io

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

7.5.3 Summary

- The `save` and `load` functions can be used to perform File I/O for `ndarray` objects.
- The `load_parameters` and `save_parameters` functions allow us to save entire sets of parameters for a network in Gluon.
- Saving the architecture has to be done in code rather than in parameters.

7.5.4 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 network architecture and parameters? What restrictions would you impose on the architecture?

7.5.5 Scan the QR Code to Discuss[98]

7.6 GPUs

In the introduction to this book 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 opportunity but it also suggests a significant need to provide such performance.

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)

In this section we begin to discuss how to harness this compute 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). You might have noticed that

[98] https://discuss.mxnet.io/t/2329

MXNet `ndarray` looks almost identical to NumPy. But there are a few crucial differences. One of the key features that differentiates MXNet from NumPy is its support for diverse hardware devices.

In MXNet, every array has a context. In fact, whenever we displayed an `ndarray` so far, it added a cryptic `@cpu(0)` notice to the output which remained unexplained so far. As we will discover, this just indicates that the computation is being executed on the CPU. 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.

In short, for complex neural networks and large-scale data, using only CPUs for computation may be inefficient. In this section, 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 CUDA[99] and 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 Nov  8 05:38:09 2019
+-----------------------------------------------------------------------------+
| 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   36C    P0    51W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:00:1C.0 Off |                    0 |
| N/A   36C    P0    52W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:00:1D.0 Off |                    0 |
| N/A   41C    P0    56W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   35C    P0    53W / 300W |      0MiB / 16130MiB |      4%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

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 the CUDA version. Assuming you have CUDA 9.0 installed, you can install the MXNet version that supports CUDA 9.0 by `pip install mxnet-cu90`. 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 flows between different devices.

[99] https://developer.nvidia.com/cuda-downloads

7.6.1 Computing Devices

MXNet can specify devices, such as CPUs and GPUs, for storage and calculation. By default, MXNet creates data in the main memory and then uses 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 graphic card and the corresponding graphic 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 through `num_gpus()`.

```
npx.num_gpus()
```

```
2
```

Now we define two convenient functions that allows us to run codes even if the requested GPUs do not exist.

```
# Saved in the d2l package for later use
def try_gpu(i=0):
    """Return gpu(i) if exists, otherwise return cpu()."""
    return npx.gpu(i) if npx.num_gpus() >= i + 1 else npx.cpu()

# Saved in the d2l package for later use
def try_all_gpus():
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    ctxes = [npx.gpu(i) for i in range(npx.num_gpus())]
    return ctxes if ctxes else [npx.cpu()]

try_gpu(), try_gpu(3), try_all_gpus()
```

```
(gpu(0), cpu(0), [gpu(0), gpu(1)])
```

7.6.2 `ndarray` and GPUs

By default, `ndarray` objects are created on the CPU. Therefore, we will see the `@cpu(0)` identifier each time we print an `ndarray`.

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

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

We can use the `context` property of `ndarray` to view the device where the `ndarray` is located. It is important to note that whenever we want to operate on multiple terms they need to be in the same context. For instance, if we sum two variables, we need to make sure that both arguments are on the same device - otherwise MXNet would not know where to store the result or even how to decide where to perform the computation.

```
x.context
```

```
cpu(0)
```

Storage on the GPU

There are several ways to store an `ndarray` on the GPU. For example, we can specify a storage device with the `ctx` parameter when creating an `ndarray`. Next, we create the `ndarray` variable `a` on `gpu(0)`. Notice that when printing `a`, the device information becomes `@gpu(0)`. The `ndarray` 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 we do not create data that exceeds the GPU memory limit.

```
x = np.ones((2, 3), ctx=try_gpu())
x
```

```
array([[1., 1., 1.],
       [1., 1., 1.]], ctx=gpu(0))
```

Assuming you have at least two GPUs, the following code will create a random array on `gpu(1)`.

```
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 $\mathbf{x} + \mathbf{y}$ we need to decide where to perform this operation. For instance, we can transfer $\mathbf{x}$ to `gpu(1)` and perform the operation there. *Do not* simply add `x + 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.

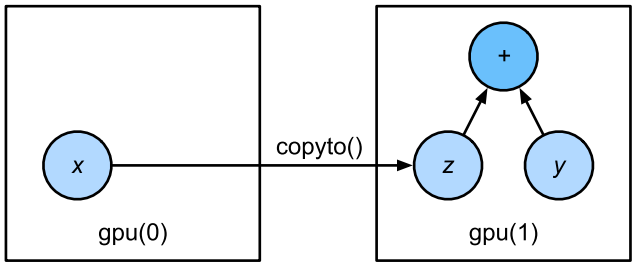


Fig. 7.6.1: Copyto copies arrays to the target device

`copyto` copies the data to another device such that we can add them. Since $\mathbf{y}$ lives on the second GPU we need to move $\mathbf{x}$ there before we can add the two.

```
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 is on the same GPU (both **z** and **y** are), we can add them up. In such cases MXNet places the result on the same device as its constituents. In our case that is `@gpu(1)`.

```
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 (gpu(1)). What happens if we 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 only want to make a copy if the variables currently lives on different contexts. In these cases, we can call `as_in_context()`. If the variable is already the specified context then this is a no-op. In fact, unless you specifically want to make a copy, `as_in_context()` is the method of choice.

```
z = x.as_in_context(try_gpu(1))
z
```

```
array([[1., 1., 1.],
       [1., 1., 1.]], ctx=gpu(1))
```

It is important to note that, if the `context` of the source variable and the target variable are consistent, then the `as_in_context` function causes the target variable and the source variable to share the memory of the source variable.

```
y.as_in_context(try_gpu(1)) is y
```

```
False
```

The `copyto` function always creates new memory for the target variable.

```
y.copyto(try_gpu(1)) is y
```

```
False
```

Watch Out

People use GPUs to do machine learning because they expect them to be fast. But transferring variables between contexts is slow. So we want you to be 100% certain that you want to do something slow before we let you do it. If MXNet 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, 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.

Lastly, when we print `ndarray`s or convert `ndarray`s to the NumPy format, if the data is not in main memory, MXNet 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 which makes everything wait for Python to complete.

7.6.3 Gluon and GPUs

Similarly, Gluon's model can specify devices through the `ctx` parameter during initialization. The following code initializes the model parameters on the GPU (we will see many more examples of how to run models on GPUs in the following, simply since they will become somewhat more compute intensive).

```
net = nn.Sequential()
net.add(nn.Dense(1))
net.initialize(ctx=try_gpu())
```

When the input is an `ndarray` on the GPU, Gluon 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()
```

```
array([[0.0068339 , 0.01299825, 0.0301265 ]], 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 we will see several such examples.

7.6.4 Summary

- MXNet can specify devices for storage and calculation, such as CPU or GPU. By default, MXNet creates data in the main memory and then uses the CPU to calculate it.
- MXNet 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 commandline (or logging it in a NumPy array) 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.

7.6.5 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 matrix norm $\mathrm{tr} MM^\top$ 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).

7.6.6 Scan the QR Code to Discuss[100]

[100] https://discuss.mxnet.io/t/2330

CHAPTER

EIGHT

CONVOLUTIONAL NEURAL NETWORKS

In several of our previous examples, we have already come up against image data, which consist of pixels arranged in a 2D grid. Depending on whether we are looking at a black and white or color image, we might have either one or multiple numerical values correspnding to each pixel location. Until now, we have dealt with this rich structure in the least satisfying possible way. We simply threw away this spatial structure by flattening each image into a 1D vector, and fed it into a fully-connected network. These networks are invariant to the order of their inputs. We will get qualitatively identical results out of a multilayer perceptron whether we preserve the original order of our features or if we permute the columns of our design matrix before learning the parameters. Ideally, we would find a way to leverage our prior knowledge that nearby pixels are more related to each other.

In this chapter, we introduce convolutional neural networks (CNNs), a powerful family of neural networks that were designed for precisely this purpose. CNN-based network *architecures* now dominate the field of computer vision to such an extent that hardly anyone these days would develop a commerical application or enter a competition related to image recognition, object detection, or semantic segmentation, without basing their approach on them.

Modern 'convnets', as they are often called owe their design to inspirations from biology, group theory, and a healthy dose of experimental tinkering. In addition to their strong predictive performance, convolutional neural networks tend to be computationally efficient, both because they tend to require fewer parameters than dense architectures and also because convolutions are easy to parralelize across GPU cores. As a result, researchers have sought to apply convnets whenever possible, and increasingly they have emerged as credible competitors even on tasks with 1D sequence structure, such as audio, text, and time series analysis, where recurrent neural networks (introduced in the next chapter) 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 operationgs that comprise the backbone of all modern 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* (also called *filters*) 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 of the recent popular neural networks whose designs are representative of most of the techniques commonly used to design modern convolutional neural networks.

8.1 From Dense Layers to Convolutions

The models that we have discussed so far are fine options if you are dealing with *tabular* data. By *tabular* we mean that the data consists of rows corresponding to examples and columns corresponding to features. With tabular data, we might anticipate that pattern we seek could require modeling interactions among the features, but do not assume anything a priori about which features are related to each other or in what way.

Sometimes we truly may not have any knowledge to guide the construction of more cleverly-organized architectures. and in thise cases, a multilayer perceptron is often the best that we can do. However, once we start dealing with high-dimensional perceptual data, these *structure-less* netwroks 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 sets of high-quality 1-megapixel photographs. This means that the input into a network has *1 million dimensions*. Even an aggressive reduction to *1,000 hidden dimensions* would require a *dense* (fully-connected) layer to support 10^9 parameters. Unless we have an extremely large dataset (perhaps billions?), lots of GPUs, a talent for extreme distributed optimization, and an extraordinary ammount of patience, learning the parameters of this network may turn out to be impossible.

A careful reader might object to this argument on the basis that 1 megapixel resolution may not be necessary. However, while you could get away with 100,000 pixels, we grossly underestimated the number of hidden nodes that it typically takes to learn good hidden representations of images. Learning a binary classifier with so many parameters might seem to require that we collect an enormous dataset, perhaps comparable to the number of dogs and cats on the planet. And yet Yet both humans and computers are able to distinguish cats from dogs quite well, seemingly contradicting these conclusions. That is because images exhibit rich structure that is typically exploited by humans and machine learning models alike.

8.1.1 Invariances

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 shouldn't in the image. Ideally we could learn a system that would somehow exploit this knowledge. Pigs usually do not fly and planes usually do not swim. Nonetheless, we could still recognize a flying pig were one to appear. This ideas is taken to an extreme in the children's game 'Where's Waldo', an example is shown in Fig. 8.1.1. The game consists of a number of chaotic scenes bursting with activity and 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 confounders.

Back to images, the intuitions we have been discussing could be made more concrete yielding a few key principles for building neural networks for computer vision:

1. Our vision systems should, in some sense, respond similary to the same object regardless of where it appears in the image (Translation Invariance)
2. Our visions systems should, in some sense, focus on local regions, without regard for what else is happening in the image at greater distances. (Locality)

Let us see how this translates into mathematics.

8.1.2 Constraining the MLP

To start off let us consider what an MLP would look like with $h \times w$ images as inputs (represented as matrices in math, and as 2D arrays in code), and hidden representations similarly organized as $h \times w$ matrices / 2D arrays. Let $x[i, j]$ and $h[i, j]$ denote pixel location (i, j) in an image and hidden representation, respectively. Consequently, to have each of the hw hidden nodes receive input from each of the hw inputs, we would switch from using weight matrices (as we did previously in MLPs) to representing our parameters as four-dimensional weight tensors.

We could formally express this dense layer as follows:

$$h[i, j] = u[i, j] + \sum_{k,l} W[i, j, k, l] \cdot x[k, l] = u[i, j] + \sum_{a,b} V[i, j, a, b] \cdot x[i + a, j + b] \tag{8.1.1}$$

The switch from W to V is entirely cosmetic (for now) since there is a one-to-one correspondence between coefficients in both tensors. We simply re-index the subscripts (k, l) such that $k = i + a$ and $l = j + b$. In other words, we set $V[i, j, a, b] = W[i, j, i + a, j + b]$. The indices a, b run over both positive and negative offsets, covering the entire image. For any given location (i, j) in the hidden layer $h[i, j]$, we compute its value by summing over pixels in x, centered around (i, j) and weighted by $V[i, j, a, b]$.

Now let us invoke the first principle we established above—*translation invariance*. This implies that a shift in the inputs x should simply lead to a shift in the activations h. This is only possible if V and u do not actually depend on (i, j), i.e.,

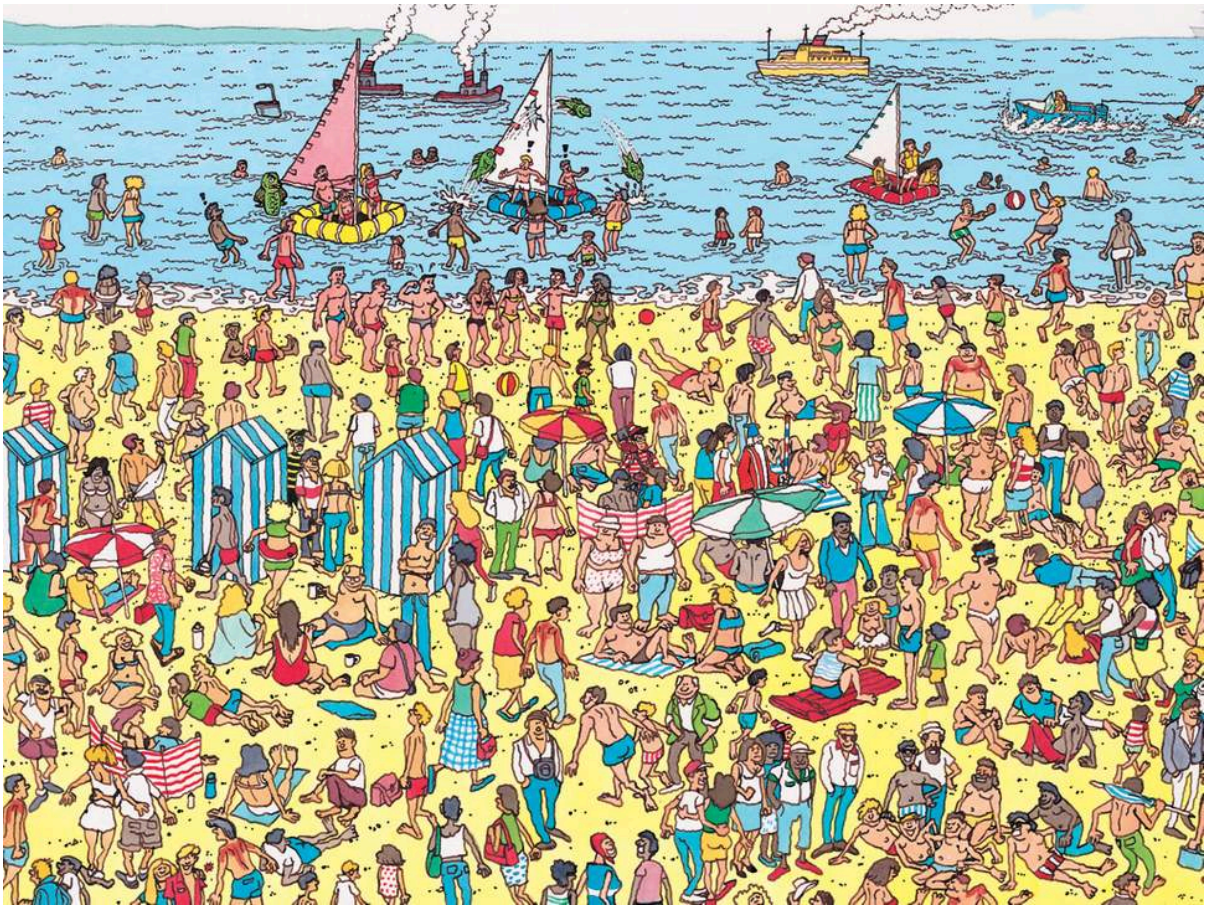


Fig. 8.1.1: Image via Walker Books

we have $V[i, j, a, b] = V[a, b]$ and u is a constant. As a result we can simplify the definition for h.

$$h[i, j] = u + \sum_{a,b} V[a, b] \cdot x[i + a, j + b] \tag{8.1.2}$$

This is a convolution! We are effectively weighting pixels $(i + a, j + b)$ in the vicinity of (i, j) with coefficients $V[a, b]$ to obtain the value $h[i, j]$. Note that $V[a, b]$ needs many fewer coefficients than $V[i, j, a, b]$. For a 1 megapixel image it has at most 1 million coefficients. This is 1 million fewer parameters since it no longer depends on the location within the image. We have made significant progress!

Now let us invoke the second principle - *locality*. As motivated above, we believe that we shouldn't have to look very far away from (i, j) in order to glean relevant information to assess what is going on at $h[i, j]$. This means that outside some range $|a|, |b| > \Delta$, we should set $V[a, b] = 0$. Equivalently, we can rewrite $h[i, j]$ as

$$h[i, j] = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} V[a, b] \cdot x[i + a, j + b] \tag{8.1.3}$$

This, in a nutshell is the convolutional layer. When the local region (also called a *receptive field*) is small, the difference as compared to 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. The price that we pay for this drastic modification is that our features will be translation invariant and that our layer can only take local information into account. 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,

8.1.3 Convolutions

Let us briefly review why the above operation is called a *convolution*. In mathematics, the convolution between two functions, say $f, g : \mathbb{R}^d \to R$ is defined as

$$[f \circledast g](x) = \int_{\mathbb{R}^d} f(z)g(x - z)dz \tag{8.1.4}$$

That is, we measure the overlap beween f and g when both functions are shifted by x and 'flipped'. Whenever we have discrete objects, the integral turns into a sum. For instance, for vectors defined on ℓ_2, i.e., the set of square summable infinite dimensional vectors with index running over $\mathbb{Z}$ we obtain the following definition.

$$[f \circledast g](i) = \sum_a f(a)g(i - a) \tag{8.1.5}$$

For two-dimensional arrays, we have a corresponding sum with indices (i, j) for f and $(i - a, j - b)$ for g respectively. This looks similar to definition above, 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 by using $\tilde{V}[a, b] = V[-a, -b]$ to obtain $h = x \circledast \tilde{V}$. Also note that the original definition is actually a *cross correlation*. We will come back to this in the following section.

8.1.4 Waldo Revisited

Let us see what this looks like if we want to build an improved Waldo detector. The convolutional layer picks windows of a given size and weighs intensities according to the mask V. We expect that wherever the 'waldoness' is highest, we will also find a peak in the hidden layer activations.

There is just a problem with this approach: so far we blissfully ignored that images consist of 3 channels: red, green and blue. In reality, images are quite two-dimensional objects but rather as a 3rd order tensor, e.g., with shape $1024 \times 1024 \times 3$

Fig. 8.1.2: Find Waldo.

pixels. Only two of these axes concern spatial relationships, while the 3rd can be regarded as assigning a multidimensional representation *to each pixel location*.

We thus index $\mathbf{x}$ as $x[i, j, k]$. The convolutional mask has to adapt accordingly. Instead of $V[a, b]$ we now have $V[a, b, c]$.

Moreover, just as our input consists of a 3rd order tensor it turns out to be a good idea to similarly formulate our hidden representations as 3rd order tensors. In other words, rather than just having a 1D representation corresponding to each spatial location, we want to have a multidimensional hidden representations corresponding to each spatial location. We could think of the hidden representation as comprising a number of 2D grids stacked on top of each other. These are sometimes called *channels* or *feature maps*. Intuitively you might imaginee that at lower layers, some channels specialize to recognizing edges, We can take care of this by adding a fourth coordinate to V via $V[a, b, c, d]$. Putting all together we have:

$$h[i, j, k] = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} V[a, b, c, k] \cdot x[i + a, j + b, c] \tag{8.1.6}$$

This is the definition of a convolutional neural network layer. There are still many operations that we need to address. For instance, we need to figure out how to combine all the activations to a single output (e.g., whether there is a Waldo in the image). We also need to decide how to compute things efficiently, how to combine multiple layers, and whether it is a good idea to have many narrow or a few wide layers. All of this will be addressed in the remainder of the chapter.

8.1.5 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 for computation.
- Channels on input and output allows for meaningful feature analysis.

8.1.6 Exercises

1. Assume that the size of the convolution mask is $\Delta = 0$. Show that in this case the convolutional mask implements an MLP independently for each set of channels.
2. Why might translation invariance not be a good idea after all? Does it make sense for pigs to fly?
3. What happens at the boundary of an image?
4. Derive an analogous convolutional layer for audio.
5. What goes wrong when you apply the above reasoning to text? Hint - what is the structure of language?
6. Prove that $f \circledast g = g \circledast f$.

8.1.7 Scan the QR Code to Discuss[101]

8.2 Convolutions for Images

Now that we understand how convolutional layers work in theory, we are ready to see how this works in practice. Since we have motivated convolutional neural networks by their applicability to image data, we will stick with image data in our examples, and begin by revisiting the convolutional layer that we introduced in the previous section. We note that strictly speaking, *convolutional* layers are a slight misnomer, since the operations are typically expressed as cross correlations.

8.2.1 The Cross-Correlation Operator

In a convolutional layer, an input array and a correlation kernel array are combined to produce an output array through a cross-correlation operation. Let us see how this works for two dimensions. In our example, the input is a two-dimensional array with a height of 3 and width of 3. We mark the shape of the array as 3×3 or (3, 3). The height and width of the kernel array are both 2. Common names for this array in the deep learning research community include *kernel* and *filter*. The shape of the kernel window (also known as the convolution window) is given precisely by the height and width of the kernel (here it is 2×2).

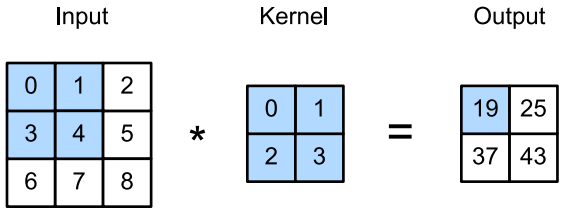


Fig. 8.2.1: Two-dimensional cross-correlation operation. The shaded portions are the first output element and the input and kernel array elements used in its computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

[101] https://discuss.mxnet.io/t/2348

In the two-dimensional cross-correlation operation, we begin with the convolution window positioned at the top-left corner of the input array and slide it across the input array, both from left to right and top to bottom. When the convolution window slides to a certain position, the input subarray contained in that window and the kernel array are multiplied (elementwise) and the resulting array is summed up yielding a single scalar value. This result if precisely the value of the output array at the corresponding location. Here, the output array has a height of 2 and width of 2 and the four elements are derived from the two-dimensional cross-correlation operation:

$$\begin{aligned}0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,\\ 1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,\\ 3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,\\ 4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.\end{aligned} \tag{8.2.1}$$

Note that along each axi, the output is slightly *smaller* than the input. Because the kernel has a width greater than one, and we can only computer the cross-correlation for locations where the kernel fits wholly within the image, the output size is given by the input size $H \times W$ minus the size of the convolutional kernel $h \times w$ via $(H - h + 1) \times (W - w + 1)$. This is the case since we need enough space to 'shift' the convolutional kernel across the image (later we will see how to keep the size unchanged by padding the image with zeros around its boundary such that there is enough space to shift the kernel). Next, we implement the above process in the `corr2d` function. It accepts the input array `X` with the kernel array `K` and outputs the array `Y`.

```
from mxnet import autograd, np, npx
from mxnet.gluon import nn
npx.set_np()

# Saved in the d2l package for later use
def corr2d(X, K):
    """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 array `X` and the kernel array `K` from the figure above to validate the output of the above implementations of the two-dimensional cross-correlation operation.

```
X = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
K = np.array([[0, 1], [2, 3]])
corr2d(X, K)
```

```
array([[19., 25.],
       [37., 43.]])
```

8.2.2 Convolutional Layers

A convolutional layer cross-correlates the input and kernels and adds a scalar bias to produce an output. The parameters of the convolutional layer are precisely the values that constitute the kernel and the scalar bias. When training the 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 computation function `forward` calls the `corr2d` function and adds the bias. As with $h \times w$ cross-correlation we also refer to convolutional layers as $h \times w$ convolutions.

```
class Conv2D(nn.Block):
    def __init__(self, kernel_size, **kwargs):
        super(Conv2D, self).__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()
```

8.2.3 Object Edge Detection in Images

Let us look at 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 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, -1]])
```

Enter `X` and our designed kernel `K` to perform the cross-correlation operations. As you can see, we will detect 1 for the edge from white to black and -1 for the edge from black to white. The rest of the outputs are 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.]])
```

Let us 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.],
```

(continues on next page)

(continued from previous page)

```
       [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.]])
```

8.2.4 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 array. Next, in each iteration, we will use the squared error to compare `Y` and the output of the convolutional layer, then calculate the gradient to update the weight. For the sake of simplicity, in this convolutional layer, we will ignores the bias.

We previously constructed the `Conv2D` class. However, since we used single-element assignments, Gluon has some trouble finding the gradient. Instead, we use the built-in `Conv2D` class provided by Gluon below.

```
# Construct a convolutional layer with 1 output channel
# (channels will be introduced in the following section)
# and a kernel array shape of (1, 2)
conv2d = nn.Conv2D(1, kernel_size=(1, 2))
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)

for i in range(10):
    with autograd.record():
        Y_hat = conv2d(X)
        l = (Y_hat - Y) ** 2
    l.backward()
    # For the sake of simplicity, we ignore the bias here
    conv2d.weight.data()[:] -= 3e-2 * conv2d.weight.grad()
    if (i + 1) % 2 == 0:
        print('batch %d, loss %.3f' % (i + 1, l.sum()))
```

```
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
```

As you can see, the error has dropped to a small value after 10 iterations. Now we will take a look at the kernel array we learned.

```
conv2d.weight.data().reshape(1, 2)
```

```
array([[ 0.9895   , -0.9873705]])
```

Indeed, the learned kernel array is remarkably close to the kernel array `K` we defined earlier.

8.2.5 Cross-correlation and Convolution

Recall the observation from the previous section that cross-correlation and convolution are equivalent. In the figure above it is easy to see this correspondence. Simply flip the kernel from the bottom left to the top right. In this case the indexing in the sum is reverted, yet the same result can 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.

8.2.6 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 through data.

8.2.7 Exercises

1. Construct an image `X` with diagonal edges.
    - What happens if you apply the kernel `K` to it?
    - What happens if you transpose `X`?
    - 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 arrays?
4. Design some kernels manually.
    - What is the form of a kernel for the second derivative?
    - What is the kernel for the Laplace operator?
    - What is the kernel for an integral?
    - What is the minimum size of a kernel to obtain a derivative of degree d?

8.2.8 Scan the QR Code to Discuss[102]

[102] https://discuss.mxnet.io/t/2349

8.3 Padding and Stride

In the previous example, our input had a height and width of 3 and a convolution kernel with a height and width of 2, yielding an output with a height and a width of 2. In general, assuming the input shape is $n_h \times n_w$ and the convolution kernel window shape is $k_h \times k_w$, then the output shape will be

$$(n_h - k_h + 1) \times (n_w - k_w + 1). \tag{8.3.1}$$

Therefore, the output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel window.

In several cases we might want to incorporate particular techniques—padding and strides—regarding the size of the output:

- In general, since kernels generally have width and height greater than 1, that means that after applying many successive convolutions, we will wind up with an output that is much smaller than our input. If we start with a 240x240 pixel image, 10 layers of 5x5 convolutions reduce the image to 200x200 pixels, slicing off 30% of the image and with it obliterating any interesting information on the boundaries of the original image. *Padding* handles this issue.
- In some cases, we want to reduce the resolution drastically if say we find our original input resolution to be unweildy. *Strides* can help in these instances.

8.3.1 Padding

As described above, one tricky issue when applying convolutional layers is that losing pixels on the permimeter 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 0. In the figure below, we pad a 3×5 input, increasing its size to 5×7. The corresponding output then increases to a 4×6 matrix.

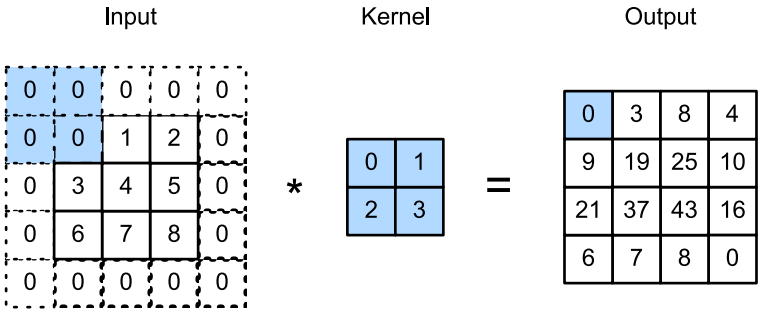


Fig. 8.3.1: Two-dimensional cross-correlation with padding. The shaded portions are the input and kernel array elements used by the first output element: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

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

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1), \tag{8.3.2}$$

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.

Convolutional neural networks commonly use convolutional 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 array `X`, when the kernels size is odd and the number of padding rows and columns on all sides are the same, producing an output with the have 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()
    # (1,1) indicates that the batch size and the number of channels
    # (described in later chapters) are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Exclude the first two dimensions that do not interest us: batch and
    # channel
    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 both sides of the height and width are 2 and 1,
# respectively
```

(continues on next page)

(continued from previous page)

```
conv2d = nn.Conv2D(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

8.3.2 Stride

When computing the cross-correlation, we start with the convolution window at the top-left corner of the input array, and then slide it over all locations both down and to the right. In previous examples, we default to sliding one pixel at a time. However, sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one pixel 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. The figure below shows a two-dimensional cross-correlation operation with a stride of 3 vertically and 2 horizontally. We can see that when the second element of the first column is output, 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 output. When the convolution window slides three 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).

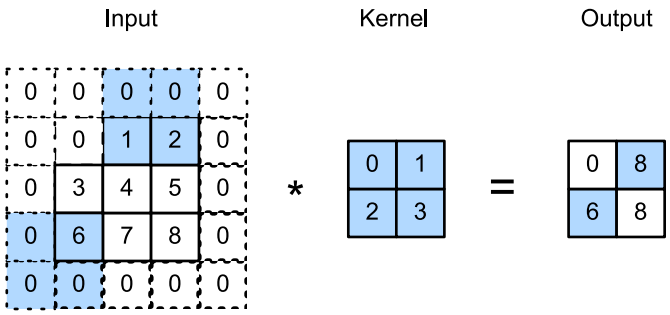


Fig. 8.3.2: Cross-correlation with strides of 3 and 2 for height and width respectively. The shaded portions are the output element and the input and core array elements used in its computation: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$, $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$.

In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor. \tag{8.3.3}$$

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 $s_h = s_w$.

8.3.3 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.

8.3.4 Exercises

1. For the last example in this section, use the shape calculation formula to calculate the output shape to see if it is consistent with the experimental results.
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.

8.3.5 Scan the QR Code to Discuss[103]

8.4 Multiple Input and 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), 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, convolutional kernels, and outputs each as two-dimensional arrays.

[103] https://discuss.mxnet.io/t/2350

When we add channels into the mix, our inputs and hidden representations both become three-dimensional arrays. 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.

8.4.1 Multiple Input Channels

When the input data contains 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 array of shape $k_h \times k_w$.

However, when $c_i > 1$, we need a kernel that contains an array of shape $k_h \times k_w$ *for each input channel*. Concatenating these c_i arrays 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 array of the input and the two-dimensional kernel array of the convolution kernel for each channel, adding the c_i results together (summing over the channels) to yield a two-dimensional array. This is the result of a two-dimensional cross-correlation between multi-channel input data and a *multi-input channel* convolution kernel.

In the figure below, 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 array elements used in its computation: $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$.

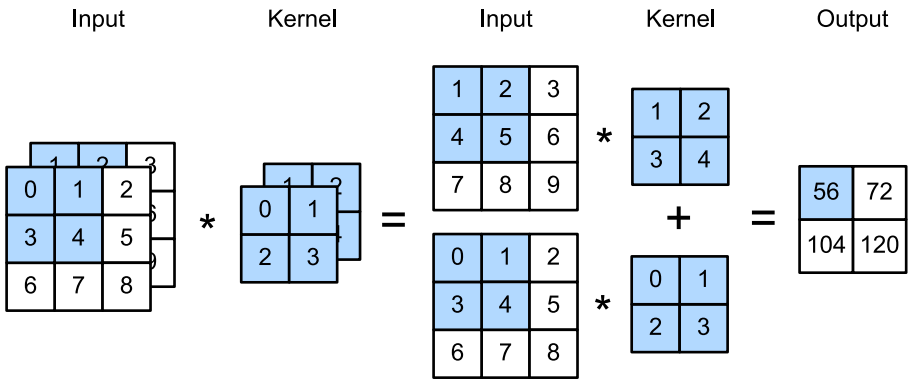


Fig. 8.4.1: Cross-correlation computation with 2 input channels. The shaded portions are the first output element as well as the input and kernel array elements used in its computation: $(1\times1+2\times2+4\times3+5\times4)+(0\times0+1\times1+3\times2+4\times3) = 56$.

To make sure we reall 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 using the `add_n` function.

```
import d2l
from mxnet import np, npx
npx.set_np()

def corr2d_multi_in(X, K):
    # First, traverse along the 0th dimension (channel dimension) of X and K.
    # Then, add them together by using * to turn the result list into a
    # positional argument of the add_n function
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

We can construct the input array `X` and the kernel array `K` corresponding to the values in the above diagram to validate the output of the cross-correlation operation.

```
X = np.array([[[0, 1, 2], [3, 4, 5], [6, 7, 8]],
              [[1, 2, 3], [4, 5, 6], [7, 8, 9]]])
K = np.array([[[0, 1], [2, 3]], [[1, 2], [3, 4]]])

corr2d_multi_in(X, K)
```

```
array([[ 56.,  72.],
       [104., 120.]])
```

8.4.2 Multiple Output Channels

Regardless of the number of input channels, so far we always ended up with one output channel. However, as we discussed earlier, 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 intepretations 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 array of shape $c_i \times k_h \times k_w$ for each 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 corrsponding to that output channel and takes input from all channels in the input array.

We implement a cross-correlation function to calculate the output of multiple channels as shown below.

```
def corr2d_multi_in_out(X, K):
    # Traverse along the 0th dimension of K, and each time, perform
    # cross-correlation operations with input X. All of the results are merged
    # together using the stack function
    return np.stack([corr2d_multi_in(X, k) for k in K])
```

We construct a convolution kernel with 3 output channels by concatenating the kernel array `K` with `K+1` (plus one for each element in `K`) and `K+2`.

```
K = np.stack((K, K + 1, K + 2))
K.shape
```

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

Below, we perform cross-correlation operations on the input array `X` with the kernel array `K`. Now the output contains 3 channels. The result of the first channel is consistent with the result of the previous input array `X` and the multi-input channel, single-output channel kernel.

```
corr2d_multi_in_out(X, K)
```

```
array([[[ 56.,  72.],
        [104., 120.]],

       [[ 76., 100.],
```

(continues on next page)

(continued from previous page)

```
        [148., 172.]],

       [[ 96., 128.],
        [192., 224.]]])
```

8.4.3 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.

The figure below 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 terms).

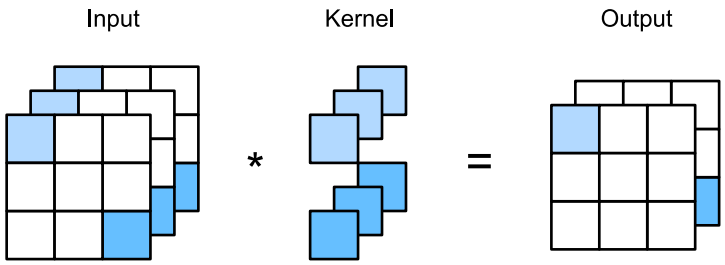


Fig. 8.4.2: The cross-correlation computation uses the 1×1 convolution kernel with 3 input channels and 2 output channels. The inputs and outputs have the same height and width.

Let us check whether this works in practice: we implement the 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)
    Y = np.dot(K, X)  # Matrix multiplication in the fully connected layer
    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 reference data.

```
X = np.random.uniform(size=(3, 3, 3))
K = np.random.uniform(size=(2, 3, 1, 1))

Y1 = corr2d_multi_in_out_1x1(X, K)
```

(continues on next page)

(continued from previous page)

```
Y2 = corr2d_multi_in_out(X, K)

np.abs(Y1 - Y2).sum() < 1e-6
```

```
array(1.)
```

8.4.4 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.

8.4.5 Exercises

1. Assume that we have two convolutional kernels of size k_1 and k_2 respectively (with no nonlinearity in between).
   - Prove that the result of the operation can be expressed by a single convolution.
   - What is the dimensionality of the equivalent single convolution?
   - Is the converse true?
2. Assume an input shape of $c_i \times h \times w$ and a convolution kernel with the 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).
   - What is the computational cost (multiplications and additions) for the forward computation?
   - What is the memory footprint?
   - What is the memory footprint for the backward computation?
   - What is the computational cost for the backward computation?
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 the convolution kernel is $k_h = k_w = 1$, what is the complexity of the forward computation?
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?

8.4.6 Scan the QR Code to Discuss[104]

[104] https://discuss.mxnet.io/t/2351

8.5 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 nodes 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 8.2), 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 and with it all the activations. 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 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.

8.5.1 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 *filter*). 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 top left of the input array and sliding across the input array 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 subarray in the window (depending on whether *max* or *average* pooling is employed).

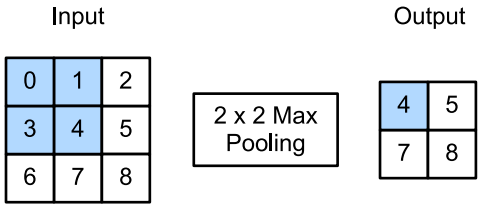


Fig. 8.5.1: Maximum pooling with a pooling window shape of 2×2. The shaded portions represent the first output element and the input element used for its computation: $\max(0, 1, 3, 4) = 4$

The output array in the figure above has a height of 2 and a width of 2. The four elements are derived from the maximum value of max:

$$\begin{aligned}\max(0, 1, 3, 4) &= 4,\\ \max(1, 2, 4, 5) &= 5,\\ \max(3, 4, 6, 7) &= 7,\\ \max(4, 5, 7, 8) &= 8.\end{aligned} \tag{8.5.1}$$

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 outputs all include `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 and width.

In the code below, we implement the forward computation of the pooling layer in the `pool2d` function. This function is similar to the `corr2d` function in Section 8.2. However, here we have no kernel, computing the output as either the max or the average of each region in the input..

```
from mxnet import np, npx
from mxnet.gluon import nn
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] = np.max(X[i: i + p_h, j: j + p_w])
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

We can construct the input array `X` in the above diagram to validate the output of the two-dimensional maximum pooling layer.

```
X = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
pool2d(X, (2, 2))
```

```
array([[4., 5.],
       [7., 8.]])
```

At the same time, we experiment with the average pooling layer.

```
pool2d(X, (2, 2), 'avg')
```

```
array([[2., 3.],
       [5., 6.]])
```

8.5.2 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 two-dimensional maximum pooling layer MaxPool2D shipped in MXNet Gluon's `nn` module. We first construct an input data of shape `(1, 1, 4, 4)`, where the first two dimensions are batch and channel.

```
X = np.arange(16).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 in the `MaxPool2D` class has the same shape as the pooling window. 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=(1, 2), strides=(2, 3))
pool2d(X)
```

```
array([[[[ 0.,  3.],
         [ 8., 11.],
         [12., 15.]]]])
```

8.5.3 Multiple Channels

When processing multi-channel input data, the pooling layer pools each input channel separately, rather than adding the inputs of each channel by channel 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 arrays `X` and `X+1` on the channel dimension to construct an input with 2 channels.

```
X = np.concatenate((X, X + 1), axis=1)
X
```

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

(continues on next page)

(continued from previous page)

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

        [[ 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.]]]])
```

8.5.4 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 functions 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 resolution.
- The pooling layer's number of output channels is the same as the number of input channels.

8.5.5 Exercises

1. Can you implement average pooling as a special case of a convolution layer? If so, do it.
2. Can you implement max 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?

8.5.6 Scan the QR Code to Discuss[105]

[105] https://discuss.mxnet.io/t/2352

8.6 Convolutional Neural Networks (LeNet)

We are now ready to put all of the tools together to deploy your first fully-functional convolutional neural network. In our first encounter with image data we applied a multilayer perceptron (Section 6.2) to pictures of clothing in the Fashion-MNIST dataset. Each image in Fashion-MNIST consisted of a two-dimensional 28×28 matrix. To make this data amenable to multilayer perceptrons which anticapte receiving inputs as one-dimensional fixed-length vectors, we first flattened each image, yielding vectors of length 784, before processing them with a series of fully-connected layers.

Now that we have introduced convolutional layers, we can keep the image in its original spatially-organized grid, processing it with a series of successive convolutional layers. Moreover, because we are using convolutional layers, we can enjoy a considerable savings in the number of parameters required.

In this section, we will introduce one of the first published convolutional neural networks whose benefit was first demonstrated by Yann Lecun, then a researcher at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images—LeNet5[106]. In the 90s, their experiments with LeNet gave the first compelling evidence that it was possible to train convolutional neural networks by backpropagation. Their model achieved outstanding results at the time (only matched by Support Vector Machines at the time) and was adopted to recognize digits for processing deposits in ATM machines. Some ATMs still runn the code that Yann and his colleague Leon Bottou wrote in the 1990s!

8.6.1 LeNet

In a rough sense, we can think LeNet as consisting of two parts: (i) a block of convolutional layers; and (ii) a block of fully-connected layers. Before getting into the weeds, let us briefly review the model in Fig. 8.6.1.

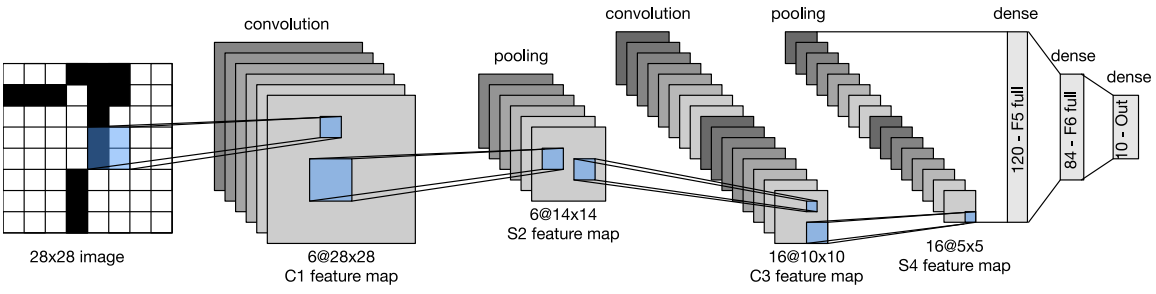


Fig. 8.6.1: Data flow in LeNet 5. The input is a handwritten digit, the output a probabilitiy over 10 possible outcomes.

The basic units in the convolutional block are a convolutional layer and a subsequent average pooling layer (note that max-pooling works better, but it had not been invented in the 90s yet). The convolutional layer is used to recognize the spatial patterns in the image, such as lines and the parts of objects, and the subsequent average pooling layer is used to

[106] http://yann.lecun.com/exdb/lenet/

reduce the dimensionality. The convolutional layer block is composed of repeated stacks of these two basic units. Each convolutional layer uses a 5×5 kernel and processes each output with a sigmoid activation function (again, note that ReLUs are now known to work more reliably, but had not been invented yet). The first convolutional layer has 6 output channels, and second convolutional layer increases channel depth further to 16.

However, coinciding with this increase in the number of channels, the height and width are shrunk considerably. Therefore, increasing the number of output channels makes the parameter sizes of the two convolutional layers similar. The two average pooling layers are of size 2×2 and take stride 2 (note that this means they are non-overlapping). In other words, the pooling layer downsamples the representation to be precisely *one quarter* the pre-pooling size.

The convolutional block emits an output with size given by (batch size, channel, height, width). Before we can pass the convolutional block's output to the fully-connected block, we must flatten each example in the minibatch. In other words, we take this 4D input and tansform it into the 2D input expected by fully-connected layers: as a reminder, the first dimension indexes the examples in the minibatch and the second gives the flat vector representation of each example. LeNet's fully-connected layer 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, you can see below that implementing it in a modern deep learning library is remarkably simple. Again, we will rely on the Sequential class.

```
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
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 the input of the shape (batch size, channel,
        # height, width) into the input of the shape (batch size,
        # channel * height * width) automatically by default
        nn.Dense(120, activation='sigmoid'),
        nn.Dense(84, activation='sigmoid'),
        nn.Dense(10))
```

As compared to the original network, we took the liberty of replacing the Gaussian activation in the last layer by a regular dense layer, which tends to be significantly more convenient to train. Other than that, this network matches the historical definition of LeNet5. Next, we feed a single-channel example of size 28×28 into the network and perform a forward computation layer by layer printing the output shape at each layer to make sure we understand what is happening here.

```
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 to the previous layer). The convolutional layer uses a kernel with a height and width of 5, which with only 2 pixels of padding in the first convolutional layer and none in the second convolutional layer leads to reductions in both height and width by 2 and 4 pixels, respectively. Moreover each pooling layer halves the height and width. However, 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 layer. Then, the fully-connected layer reduces dimensionality layer by layer, until emitting an output that matches the number of image classes.

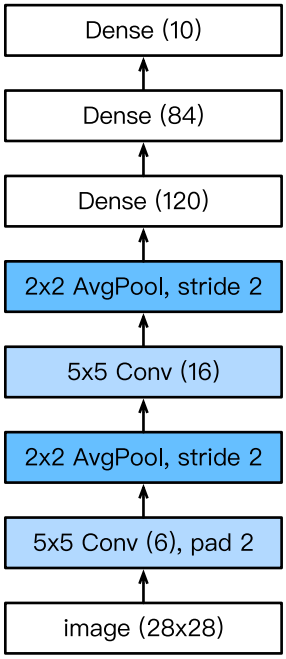


Fig. 8.6.2: Compressed notation for LeNet5

8.6.2 Data Acquisition and Training

Now that we have implemented the model, we might as well run some experiments to see what we can accomplish with the LeNet model. While it might serve nostalgia to train LeNet on the original MNIST dataset, that dataset has become too easy, with MLPs getting over 98% accuracy, so it would be hard to see the benefits of convolutional networks. Thus we will stick with Fashion-MNIST as our dataset because while it has the same shape (28×28 images), this dataset is notably more challenging.

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

While convolutional networks may have few parameters, they can still be significantly more expensive to compute than a similarly deep multilayer perceptron so 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 5.6. Since the full dataset lives on the CPU, we need to copy it to the GPU before we can compute our models. This is accomplished via the `as_in_context` function described in Section 7.6.

```
# Saved in the d2l package for later use
def evaluate_accuracy_gpu(net, data_iter, ctx=None):
```

(continues on next page)

(continued from previous page)

```
    if not ctx:  # Query the first device the first parameter is on.
        ctx = list(net.collect_params().values())[0].list_ctx()[0]
    metric = d2l.Accumulator(2)  # num_corrected_examples, num_examples
    for X, y in data_iter:
        X, y = X.as_in_context(ctx), y.as_in_context(ctx)
        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 5.6, we now need to move each batch of data to our designated context (hopefully, the GPU) prior to making the forward and backward passes.

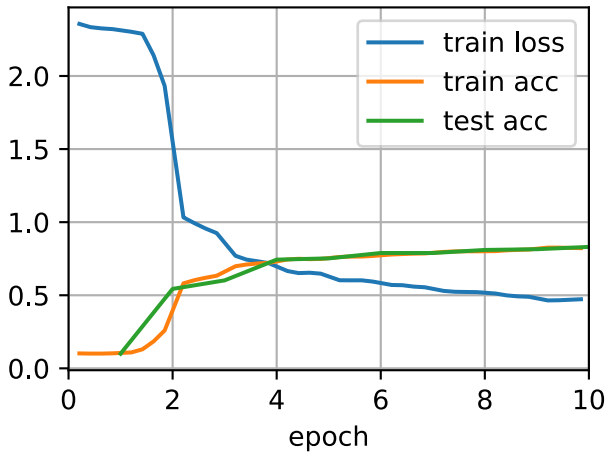
The training function `train_ch5` is also very similar to `train_ch3` defined in Section 5.6. Since we will deal with networks with tens of layers now, the function will only support Gluon models. We initialize the model parameters on the device indicated by `ctx`, this time using the Xavier initializer. The loss function and the training algorithm still use the cross-entropy loss function and minibatch stochastic gradient descent. Since each epoch takes tens of second to run, we visualize the training loss in a finer granularity.

```
# Saved in the d2l package for later use
def train_ch5(net, train_iter, test_iter, num_epochs, lr, ctx=d2l.try_gpu()):
    net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(),
                            'sgd', {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epoch', xlim=[0,num_epochs],
                            legend=['train loss','train acc','test acc'])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            # Here is the only difference compared to train_epoch_ch3
            X, y = X.as_in_context(ctx), y.as_in_context(ctx)
            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_loss, train_acc = metric[0]/metric[2], metric[1]/metric[2]
            if (i+1) % 50 == 0:
                animator.add(epoch + i/len(train_iter),
                             (train_loss, train_acc, None))
        test_acc = evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch+1, (None, None, test_acc))
    print('loss %.3f, train acc %.3f, test acc %.3f' % (
        train_loss, train_acc, test_acc))
    print('%.1f exampes/sec on %s'%(metric[2]*num_epochs/timer.sum(), ctx))
```

Now let us train the model.

```
lr, num_epochs = 0.9, 10
train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.473, train acc 0.822, test acc 0.831
56330.0 exampes/sec on gpu(0)
```



8.6.3 Summary

- A convolutional neural network (in short, ConvNet) is a network using convolutional layers.
- In a ConvNet we alternate between convolutions, nonlinearities and often also pooling operations.
- Ultimately the resolution is reduced prior to emitting an output via one (or more) dense layers.
- LeNet was the first successful deployment of such a network.

8.6.4 Exercises

1. Replace the average pooling with max pooling. What happens?
2. Try to construct a more complex network based on LeNet to improve its accuracy.
    - Adjust the convolution window size.
    - Adjust the number of output channels.
    - Adjust the activation function (ReLU?).
    - Adjust the number of convolution layers.
    - Adjust the number of fully connected layers.
    - Adjust the learning rates and other training details (initialization, epochs, etc.)
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, coats).

8.6.5 Scan the QR Code to Discuss[107]

[107] https://discuss.mxnet.io/t/2353

CHAPTER

NINE

MODERN CONVOLUTIONAL NETWORKS

Now that we understand the basics of wiring together convolutional neural networks, we will take you through a tour of modern deep learning. In this chapter, each section will correspond to a significant neural network architecture that was at some point (or currently) the base model upon which an enormous amount of research and projects were built. Each of these networks was at briefly a dominant architecture an d many were at one point winners or runners-up in the famous 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; the GoogLeNet, which makes use of networks with parallel concatenations (GoogLeNet); residual networks (ResNet) which are currently the most popular go-to architecture today, and densely connected networks (DenseNet), which are expensive to compute but have set some recent benchmarks.

9.1 Deep Convolutional Neural Networks (AlexNet)

Although convolutional neural networks 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 feasability of training convolutional networks 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 convolutional neural networks 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 SIFT[108], the Scale-Invariant Feature Transform, or SURF[109], the Speeded-Up Robust Features, or any number of other hand-tuned pipelines.
4. Dummp the resulting representations into your favorite classifier, likely a linear model or kernel method, to learn 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'd hear a very different story. The dirty truth of image recognition, they'd 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.

9.1.1 Learning Feature Representation

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[110], SURF[111], HOG[112], Bags of visual words[113] 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, (Krizhevsky et al., 2012) proposed a new variant of a convolutional neural network which achieved excellent performance in the ImageNet challenge.

Interestingly in the lowest layers of the network, the model learned featrue extractors that resembled some traditional filters. The figure below is reproduced from this paper and describes lower-level image descriptors.

Higher layers in the network might build upon these representations to represent larger structures, like eyes, noses, blades of grass, etc. 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 be separated easily.

While the ultimate breakthrough for many-layered convolutional networks 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 signficantly 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, 1,000 each from 1,000 distinct categories of objects. The researchers, led by Fei-Fei Li, who introduced this dataset leveraged Google

[108] https://en.wikipedia.org/wiki/Scale-invariant_feature_transform
[109] https://en.wikipedia.org/wiki/Speeded_up_robust_features
[110] https://en.wikipedia.org/wiki/Scale-invariant_feature_transform
[111] https://en.wikipedia.org/wiki/Speeded_up_robust_features
[112] https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients
[113] https://en.wikipedia.org/wiki/Bag-of-words_model_in_computer_vision

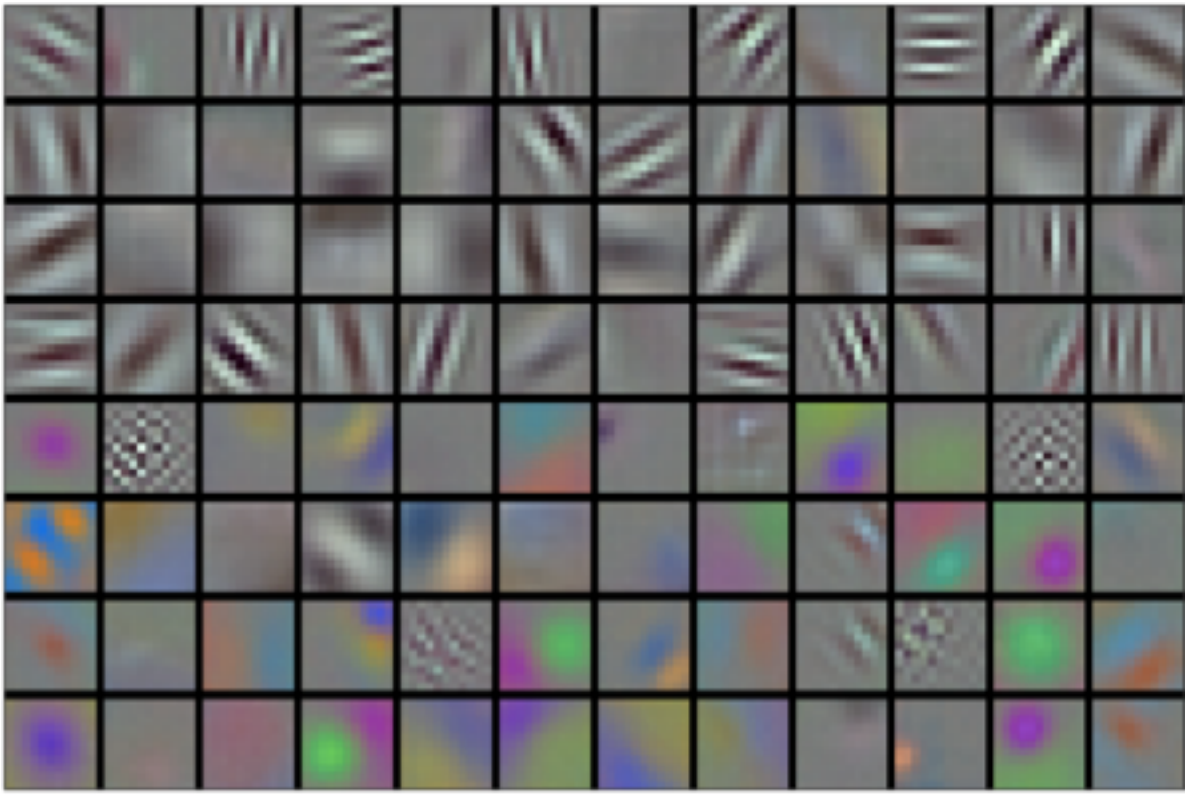


Fig. 9.1.1: Image filters learned by the first layer of AlexNet

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.

Missing Ingredient - Hardware

Deep learning models are varacious 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 90s 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 make deep learning feasible. These chips had long been develoepd for accelerating graphics processing to benefit computer games. In particular, they were optimized for high throughput 4x4 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 compute 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 MB 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, etc.), 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 100-1000 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 latest 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: firstly, power consumption tends to grow *quadratically* with clock frequency. Hence, for the power budget of a CPU core that runs 4x faster (a typical number), you can use 16 GPU cores at 1/4 the speed, which yields 16 x 1/4 = 4x the performance. Furthermore, GPU cores are much simpler (in fact, for a long time they weren't even *able* to execute general purpose code), which makes them more energy efficient. Lastly, many operations in deep learning require high memory bandwidth. Again, GPUs shine here with buses that are at least 10x as wide as many CPUs.

Back to 2012. A major breakthrough came when Alex Krizhevsky and Ilya Sutskever implemented a deep convolutional neural network 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 NIVIDA GTX 580s with 3GB of memory, they implemented fast convolutions. The code cuda-convnet[114] was good enough that for several years it was the industry standard and powered the first couple years of the deep learning boom.

9.1.2 AlexNet

AlexNet was introduced in 2012, named after Alex Krizhevsky, the first author of the breakthrough ImageNet classification paper (Krizhevsky et al., 2012). AlexNet, which employed an 8-layer convolutional neural network, won the ImageNet Large Scale Visual Recognition Challenge 2012 by a phenomenally large margin. This network proved, for the first time, that the features obtained by learning can transcend manually-design features, breaking the previous paradigm in computer vision. The architectures of AlexNet and LeNet are *very similar*, as the diagram below 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.

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 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

Second, AlexNet changed the sigmoid activation function to a simpler ReLU activation function. On the one hand, the computation of the ReLU activation function is simpler. For example, it does not have the exponentiation operation found

[114] https://code.google.com/archive/p/cuda-convnet/

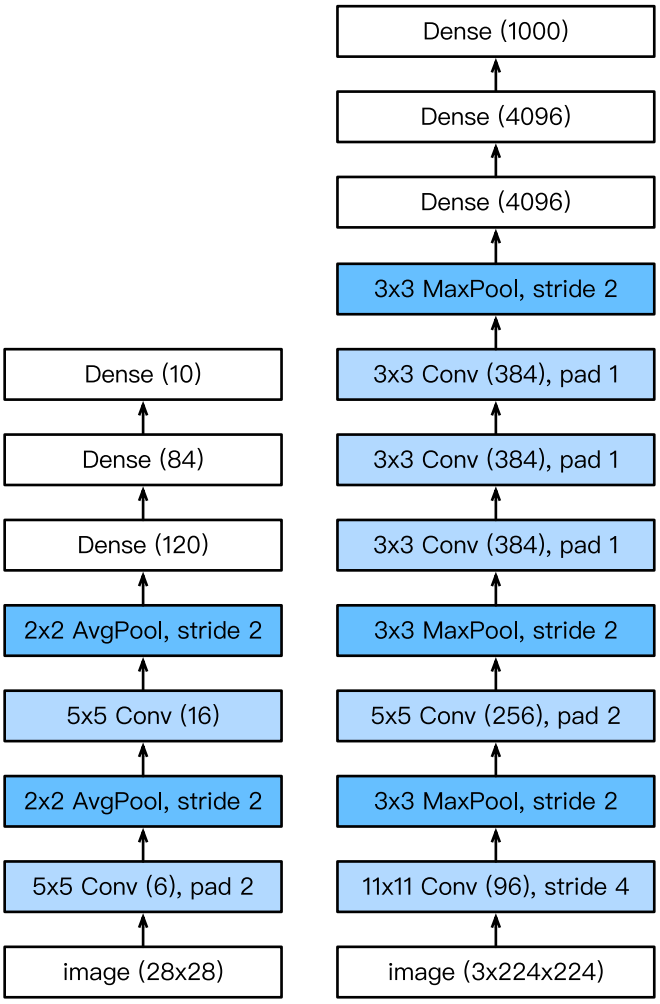


Fig. 9.1.2: LeNet (left) and AlexNet (right)

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 back propagation 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 6.6), 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 14.1.

```
import d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
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'),
        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 instance with both height and width of 224 to observe the output shape of each layer. It matches our diagram above.

```
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)
```

9.1.3 Reading Data

Although AlexNet uses 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 our images are lower resolution (28×28 pixels) than ImageNet images. To make things work, we upsample them to 244×244 (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 `load_data_fashion_mnist`.

```
batch_size = 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
```

9.1.4 Training

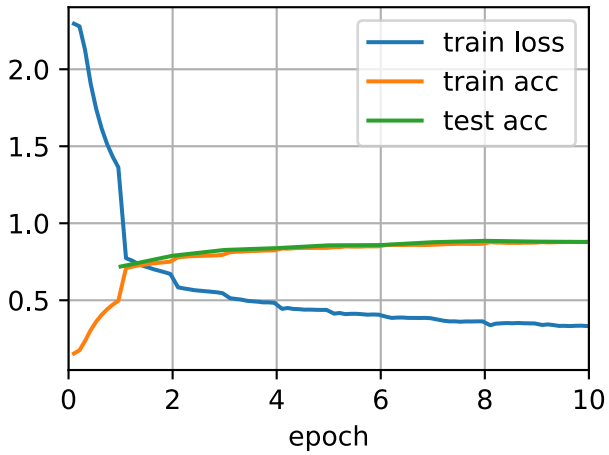
Now, we can start training AlexNet. Compared to LeNet in the previous section, the main change here is the use of a smaller learning rate and much slower training due to the deeper and wider network, the higher image resolution and the more costly convolutions.

```
lr, num_epochs = 0.01, 10
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.333, train acc 0.879, test acc 0.879
4145.6 exampes/sec on gpu(0)
```

9.1.5 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 dataset ImageNet.
- 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.



9.1.6 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.
   - Try to simplify the model to make the training faster, while ensuring that the accuracy does not drop significantly.
   - Can you 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. Rooflines
   - What is the dominant part for the memory footprint of AlexNet?
   - What is the dominant part for computation in AlexNet?
   - How about memory bandwidth when computing the results?
5. Apply dropout and ReLU to LeNet5. Does it improve? How about preprocessing?

9.1.7 Scan the QR Code to Discuss[115]

9.2 Networks Using Blocks (VGG)

While AlexNet proved that deep convolutional neural networks can achieve good results, it did not offer 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.

[115] https://discuss.mxnet.io/t/2354

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[116] (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.

9.2.1 VGG Blocks

The basic building block of classic convolutional networks is a sequence of the following layers: (i) a convolutional layer (with padding to maintain the resolution), (ii) a nonlinearity such as a ReLu, One VGG block consists of a sequence of convolutional layers, followed by a max pooling layer for spatial downsampling. In the original VGG paper (Simonyan & Zisserman, 2014), the authors employed convolutions with 3×3 kernels and 2×2 max 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`.

```
import d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
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
```

9.2.2 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 a second consisting of fully-connected layers. The convolutional portion of the net connects several `vgg_block` modules in succession. Below, the 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 requires to call the `vgg_block` function. The fully-connected module 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 layer part
```

(continues on next page)

[116] http://www.robots.ox.ac.uk/~vgg/

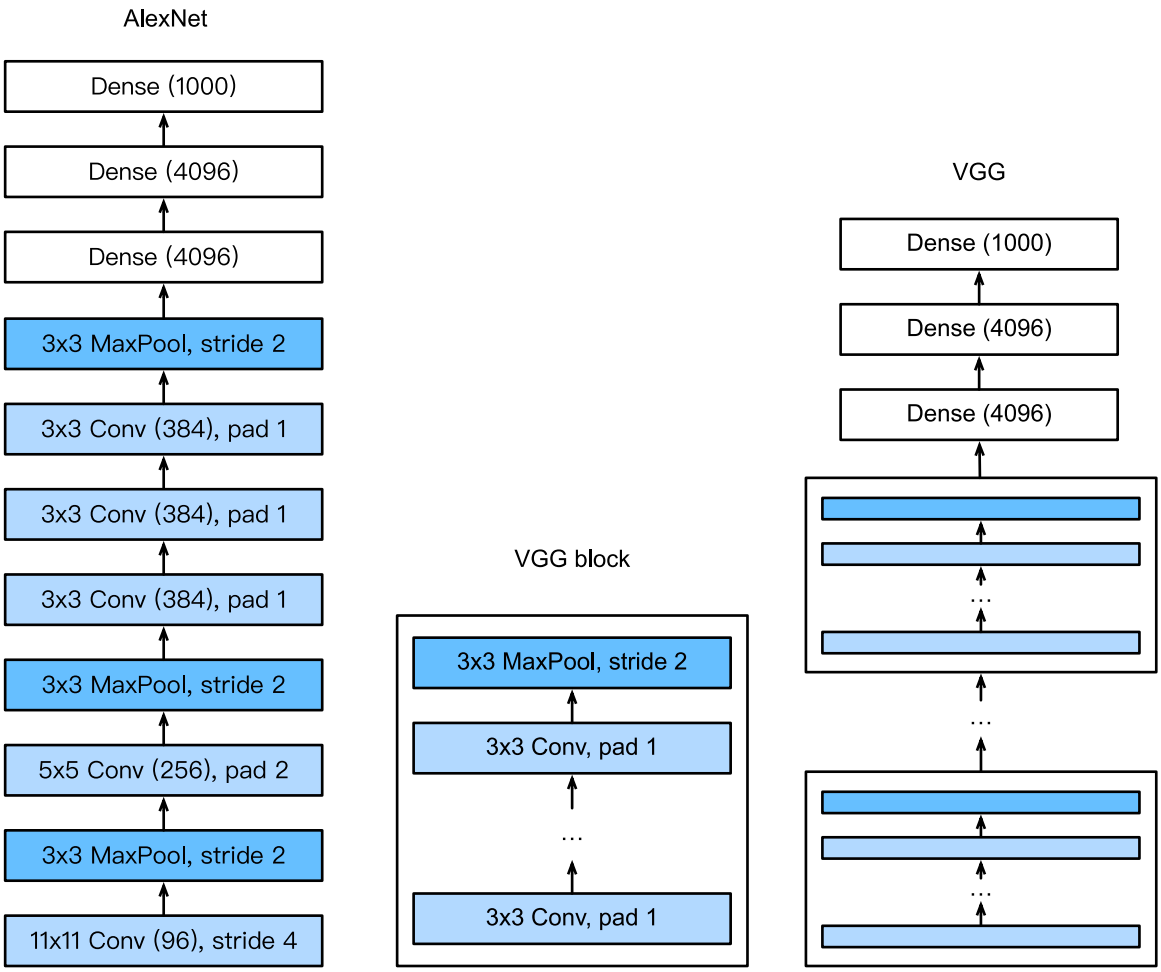


Fig. 9.2.1: Designing a network from building blocks

(continued from previous page)

```
    for (num_convs, num_channels) in conv_arch:
        net.add(vgg_block(num_convs, num_channels))
    # The fully connected layer 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 layer.

9.2.3 Model Training

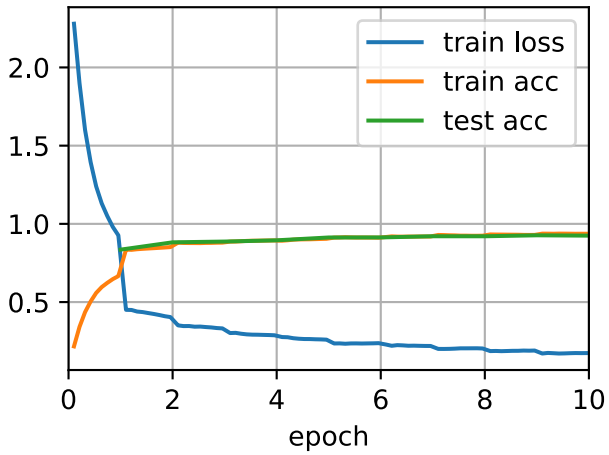
Since VGG-11 is more computationally-heavy than AlexNet we construct a network with a smaller number of channels. This is more than sufficient for training on Fashion-MNIST.

```
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 the last section.

```
lr, num_epochs, batch_size = 0.05, 10, 128,
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.174, train acc 0.937, test acc 0.925
1818.1 exampes/sec on gpu(0)
```



9.2.4 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 work 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.

9.2.5 Exercises

1. When printing out the dimensions of the layers we only saw 8 results rather than 11. Where did the remaining 3 layer informations go?
2. Compared with AlexNet, VGG is much slower in terms of computation, and it also needs more GPU memory. Try to analyze the reasons for this.
3. Try to change 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 (Simonyan & Zisserman, 2014) to construct other common models, such as VGG-16 or VGG-19.

9.2.6 Scan the QR Code to Discuss[117]

[117] https://discuss.mxnet.io/t/2355

9.3 Network in Network (NiN)

LeNet, AlexNet, and VGG all share a common design pattern: extract features exploiting *spatial* structure via a sequence of convolutions 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 in (Lin et al., 2013) based on a very simple insight—to use an MLP on the channels for each pixel separately.

9.3.1 NiN Blocks

Recall that the inputs and outputs of convolutional layers consist of four-dimensional arrays with axes corresponding to the batch, channel, height, and width. Also recall that the inputs and outputs of fully-connected layers are typically two-dimensional arrays corresponding to the batch, and features. 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 8.4) 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 the channel as equivalent to a feature. The figure below illustrates the main structural differences between NiN and AlexNet, VGG, and other networks.

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 width of the first layer is typically set by the user. The subsequent widths are fixed to 1×1.

```
import d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
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
```

9.3.2 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.

Once significant difference between NiN and AlexNet is that NiN avoids dense connections 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[118]. 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),
```

(continues on next page)

[118] https://en.wikipedia.org/wiki/Logit

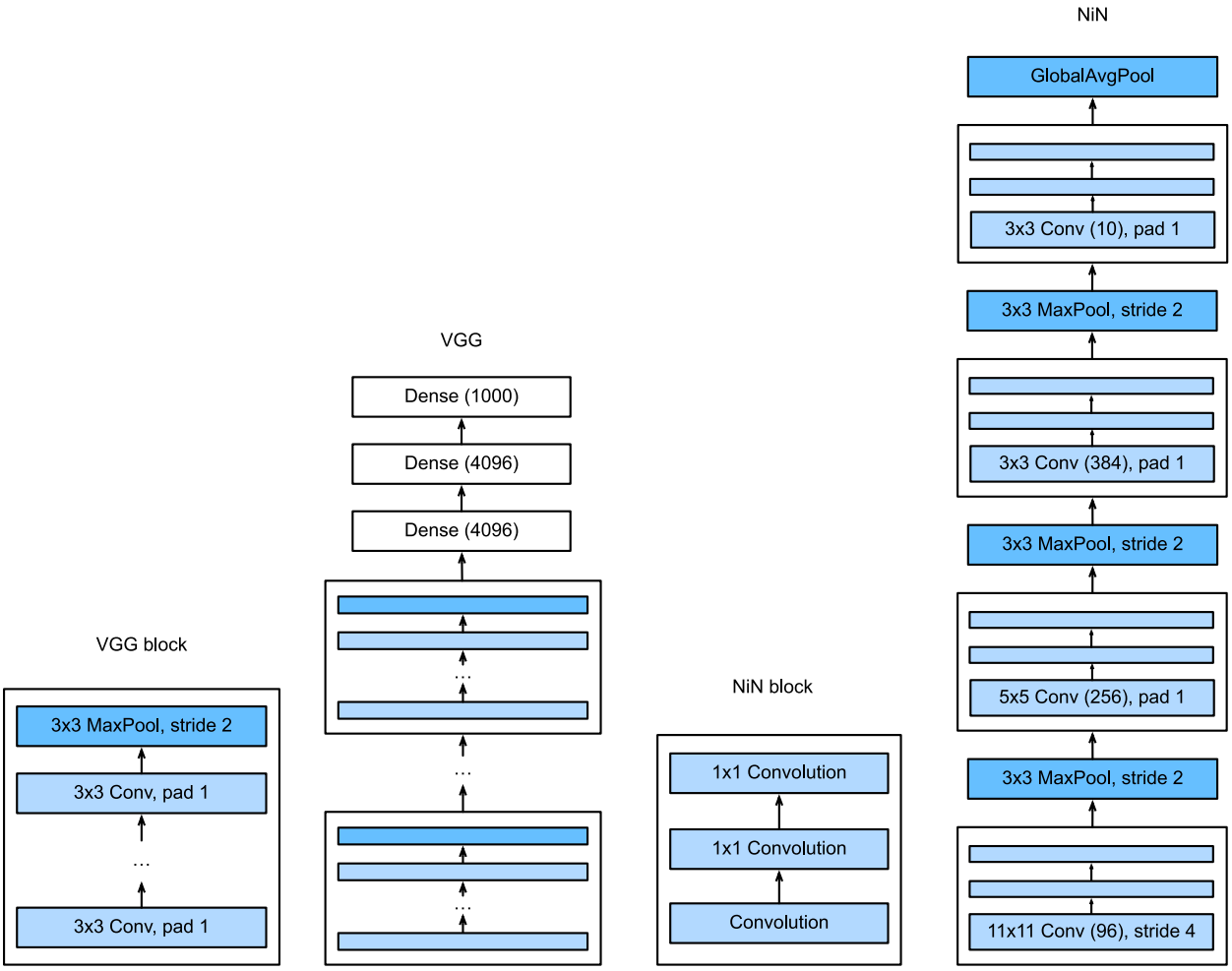


Fig. 9.3.1: The figure on the left shows the network structure of AlexNet and VGG, and the figure on the right shows the network structure of NiN.

(continued from previous page)

```
        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)
```

9.3.3 Data Acquisition and Training

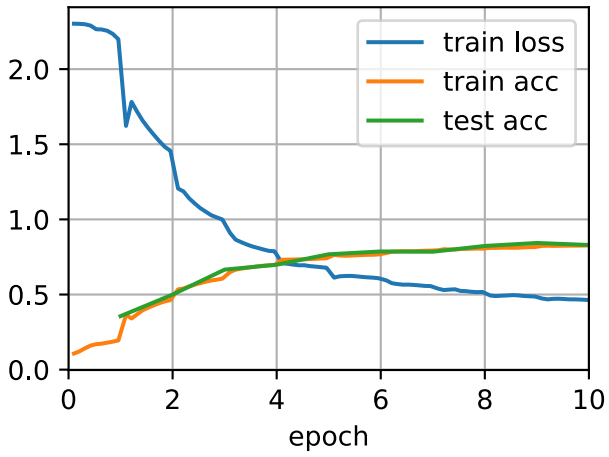
As before we use Fashion-MNIST to train the model. NiN's training is similar to that for AlexNet and VGG, but it often uses a larger learning rate.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.464, train acc 0.826, test acc 0.830
2949.1 exampes/sec on gpu(0)
```

9.3.4 Summary

- NiN uses blocks consisting of a convolutional layer and multiple 1×1 convolutional layer. This can be used within the convolutional stack to allow for more per-pixel nonlinearity.



- NiN removes the fully connected layers and replaces them with global average pooling (i.e., summing over all locations) after reducing the number of channels to the desired number of outputs (e.g., 10 for Fashion-MNIST).
- Removing the dense layers reduces overfitting. NiN has dramatically fewer parameters.
- The NiN design influenced many subsequent convolutional neural networks designs.

9.3.5 Exercises

1. Tune the hyper-parameters to improve the classification accuracy.
2. Why are there two 1×1 convolutional layers in the NiN block? Remove one of them, and then observe and analyze the experimental phenomena.
3. Calculate the resource usage for NiN
    - What is the number of parameters?
    - What is the amount of computation?
    - What is the amount of memory needed during training?
    - What is the amount of memory needed during inference?
4. What are possible problems with reducing the $384 \times 5 \times 5$ representation to a $10 \times 5 \times 5$ representation in one step?

9.3.6 Scan the QR Code to Discuss[119]

[119] https://discuss.mxnet.io/t/2356

9.4 Networks with Parallel Concatenations (GoogLeNet)

In 2014, (Szegedy et al., 2015) won the ImageNet Challenge, proposing a structure that combined the strengths of the NiN and repeated blocks paradigms. One focus of the paper was to address the question of which sized convolutional kernels are best. After all, previous popular networks employed choices as small as 1×1 and as large as 11×11. One insight in this paper was that sometimes it can be advantageous to employ a combination of variously-sized kernels. In this section, we will introduce GoogLeNet, presenting a slightly simplified version of the original model—we omit a few ad hoc features that were added to stabilize training but are unnecessary now with better training algorithms available.

9.4.1 Inception Blocks

The basic convolutional block in GoogLeNet is called an Inception block, likely named due to a quote from the movie Inception ("We Need To Go Deeper"), which launched a viral meme.

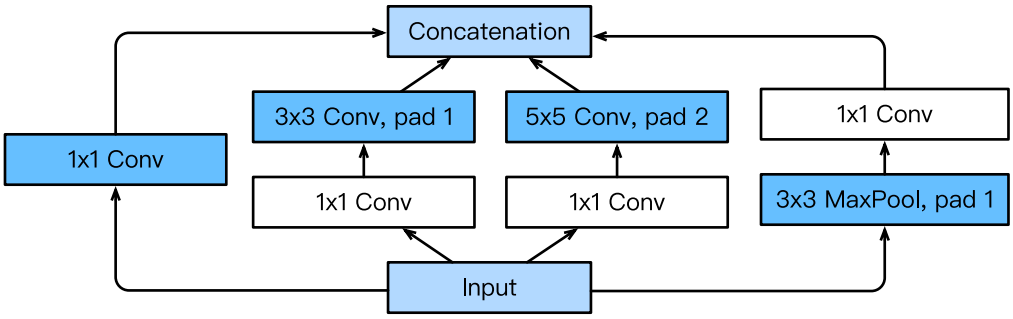


Fig. 9.4.1: Structure of the Inception block.

As depicted in the figure above, the inception block consists of four parallel paths. The first three paths use convolutional layers with window sizes of 1×1, 3×3, and 5×5 to extract information from different spatial sizes. The middle two paths perform a 1×1 convolution on the input to reduce the number of input channels, reducing the model's complexity. The fourth path uses a 3×3 maximum pooling layer, followed by a 1×1 convolutional layer to change the number of channels. The four paths all use appropriate padding to give the input and output the same height and width. Finally, the outputs along each path are concatenated along the channel dimension and comprise the block's output. The commonly-tuned parameters of the Inception block are the number of output channels per layer.

```
import d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
npx.set_np()

class Inception(nn.Block):
    # c1 - c4 are the number of output channels for each layer in the path
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Path 1 is a single 1 x 1 convolutional layer
        self.p1_1 = nn.Conv2D(c1, kernel_size=1, activation='relu')
        # Path 2 is a 1 x 1 convolutional layer followed by a 3 x 3
        # convolutional layer
        self.p2_1 = nn.Conv2D(c2[0], kernel_size=1, activation='relu')
        self.p2_2 = nn.Conv2D(c2[1], kernel_size=3, padding=1,
                              activation='relu')
        # Path 3 is a 1 x 1 convolutional layer followed by a 5 x 5
        # convolutional layer
```

(continues on next page)

(continued from previous page)

```
        self.p3_1 = nn.Conv2D(c3[0], kernel_size=1, activation='relu')
        self.p3_2 = nn.Conv2D(c3[1], kernel_size=5, padding=2,
                              activation='relu')
        # Path 4 is a 3 x 3 maximum pooling layer followed by a 1 x 1
        # convolutional layer
        self.p4_1 = nn.MaxPool2D(pool_size=3, strides=1, padding=1)
        self.p4_2 = nn.Conv2D(c4, kernel_size=1, activation='relu')

    def forward(self, x):
        p1 = self.p1_1(x)
        p2 = self.p2_2(self.p2_1(x))
        p3 = self.p3_2(self.p3_1(x))
        p4 = self.p4_2(self.p4_1(x))
        # Concatenate the outputs on the channel dimension
        return np.concatenate((p1, p2, p3, p4), axis=1)
```

To gain some intuition for why this network works so well, consider the combination of the filters. They explore the image in varying ranges. This means that details at different extents can be recognized efficiently by different filters. At the same time, we can allocate different amounts of parameters for different ranges (e.g., more for short range but not ignore the long range entirely).

9.4.2 GoogLeNet Model

GoogLeNet uses a stack of a total of 9 inception blocks and global average pooling to generate its estimates. Maximum pooling between inception blocks reduced the dimensionality. The first part is identical to AlexNet and LeNet, the stack of blocks is inherited from VGG and the global average pooling avoids a stack of fully-connected layers at the end. The architecture is depicted below.

We can now implement GoogLeNet piece by piece. The first component uses a 64-channel 7×7 convolutional layer.

```
b1 = nn.Sequential()
b1.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3, activation='relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The second component uses two convolutional layers: first, a 64-channel 1×1 convolutional layer, then a 3×3 convolutional layer that triples the number of channels. This corresponds to the second path in the Inception block.

```
b2 = nn.Sequential()
b2.add(nn.Conv2D(64, kernel_size=1, activation='relu'),
       nn.Conv2D(192, kernel_size=3, padding=1, activation='relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The third component connects two complete Inception blocks in series. The number of output channels of the first Inception block is $64 + 128 + 32 + 32 = 256$, and the ratio to the output channels of the four paths is $64 : 128 : 32 : 32 = 2 : 4 : 1 : 1$. The second and third paths first reduce the number of input channels to $96/192 = 1/2$ and $16/192 = 1/12$, respectively, and then connect the second convolutional layer. The number of output channels of the second Inception block is increased to $128 + 192 + 96 + 64 = 480$, and the ratio to the number of output channels per path is $128 : 192 : 96 : 64 = 4 : 6 : 3 : 2$. The second and third paths first reduce the number of input channels to $128/256 = 1/2$ and $32/256 = 1/8$, respectively.

```
b3 = nn.Sequential()
b3.add(Inception(64, (96, 128), (16, 32), 32),
```

(continues on next page)

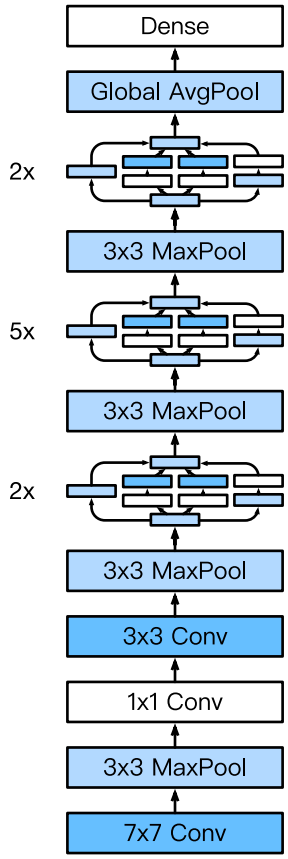


Fig. 9.4.2: Full GoogLeNet Model

(continued from previous page)

```
        Inception(128, (128, 192), (32, 96), 64),
        nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The fourth block is more complicated. It connects five Inception blocks in series, and they have $192+208+48+64 = 512$, $160+224+64+64 = 512$, $128+256+64+64 = 512$, $112+288+64+64 = 528$, and $256+320+128+128 = 832$ output channels, respectively. The number of channels assigned to these paths is similar to that in the third module: the second path with the 3×3 convolutional layer outputs the largest number of channels, followed by the first path with only the 1×1 convolutional layer, the third path with the 5×5 convolutional layer, and the fourth path with the 3×3 maximum pooling layer. The second and third paths will first reduce the number of channels according the ratio. These ratios are slightly different in different Inception blocks.

```
b4 = nn.Sequential()
b4.add(Inception(192, (96, 208), (16, 48), 64),
       Inception(160, (112, 224), (24, 64), 64),
       Inception(128, (128, 256), (24, 64), 64),
       Inception(112, (144, 288), (32, 64), 64),
       Inception(256, (160, 320), (32, 128), 128),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The fifth block has two Inception blocks with $256 + 320 + 128 + 128 = 832$ and $384 + 384 + 128 + 128 = 1024$ output channels. The number of channels assigned to each path is the same as that in the third and fourth modules, but differs in specific values. It should be noted that the fifth block is followed by the output layer. This block uses the global average pooling layer to change the height and width of each channel to 1, just as in NiN. Finally, we turn the output into a two-dimensional array followed by a fully-connected layer whose number of outputs is the number of label classes.

```
b5 = nn.Sequential()
b5.add(Inception(256, (160, 320), (32, 128), 128),
       Inception(384, (192, 384), (48, 128), 128),
       nn.GlobalAvgPool2D())

net = nn.Sequential()
net.add(b1, b2, b3, b4, b5, nn.Dense(10))
```

The GoogLeNet model is computationally complex, so it is not as easy to modify the number of channels as in VGG. To have a reasonable training time on Fashion-MNIST, we reduce the input height and width from 224 to 96. This simplifies the computation. The changes in the shape of the output between the various modules is demonstrated below.

```
X = np.random.uniform(size=(1, 1, 96, 96))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

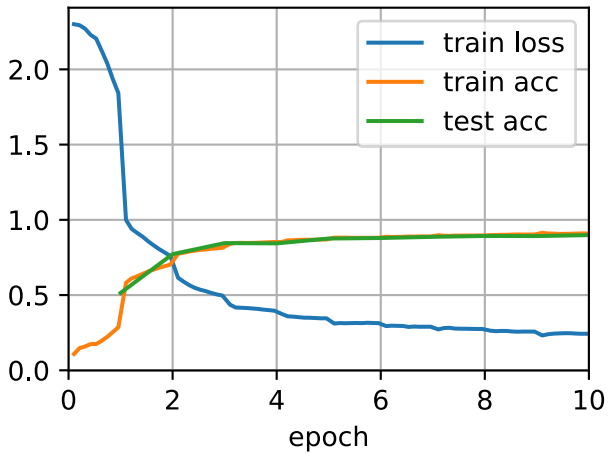
```
sequential0 output shape:    (1, 64, 24, 24)
sequential1 output shape:    (1, 192, 12, 12)
sequential2 output shape:    (1, 480, 6, 6)
sequential3 output shape:    (1, 832, 3, 3)
sequential4 output shape:    (1, 1024, 1, 1)
dense0 output shape:         (1, 10)
```

9.4.3 Data Acquisition and Training

As before, we train our model using the Fashion-MNIST dataset. We transform it to 96×96 pixel resolution before invoking the training procedure.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.242, train acc 0.910, test acc 0.898
2526.8 exampes/sec on gpu(0)
```



9.4.4 Summary

- The Inception block is equivalent to a subnetwork with four paths. It extracts information in parallel through convolutional layers of different window shapes and maximum pooling layers. 1×1 convolutions reduce channel dimensionality on a per-pixel level. Max-pooling reduces the resolution.
- GoogLeNet connects multiple well-designed Inception blocks with other layers in series. The ratio of the number of channels assigned in the Inception block is obtained through a large number of experiments on the ImageNet dataset.
- GoogLeNet, as well as its succeeding versions, was one of the most efficient models on ImageNet, providing similar test accuracy with lower computational complexity.

9.4.5 Exercises

1. There are several iterations of GoogLeNet. Try to implement and run them. Some of them include the following:
   - Add a batch normalization layer (Ioffe & Szegedy, 2015), as described later in Section 9.5.
   - Make adjustments to the Inception block (Szegedy et al., 2016).
   - Use "label smoothing" for model regularization (Szegedy et al., 2016).
   - Include it in the residual connection (Szegedy et al., 2017), as described later in Section 9.6.
2. What is the minimum image size for GoogLeNet to work?

3. Compare the model parameter sizes of AlexNet, VGG, and NiN with GoogLeNet. How do the latter two network architectures significantly reduce the model parameter size?
4. Why do we need a large range convolution initially?

9.4.6 Scan the QR Code to Discuss[120]

9.5 Batch Normalization

Training deep neural nets is difficult. And getting them to converge in a reasonable amount of time can be tricky.
In this section, we describe batch normalization (BN) (Ioffe & Szegedy, 2015), a popular and effective technique that consistently accelerates the convergence of deep nets. Together with residual blocks—covered in Section 9.6—BN has made it possible for practitioners to routinely train networks with over 100 layers.

9.5.1 Training Deep Networks

To motivate batch normalization, let us review a few practical challenges that arise when training ML models and neural nets in particular.

1. Choices regarding data preprocessing often make an enormous difference in the final results. Recall our application of multilayer perceptrons to predicting house prices (Section 6.10). Our first step when working with real data was to standardize our input features to each have a mean of *zero* and variance of *one*. Intuitively, this standardization plays nicely with our optimizers because it puts the parameters are a-priori at a similar scale.
2. For a typical MLP or CNN, as we train, the activations in intermediate layers may take values with widely varying magnitudes—both along the layers from the input to the output, across nodes in the same layer, and over time due to our updates to the model's parameters. The inventors of batch normalization postulated informally that this drift in the distribution of activations could hamper the convergence of the network. Intuitively, we might conjecture that if one layer has activation values that are 100x that of another layer, this might necessitate compensatory adjustments in the learning rates.
3. Deeper networks are complex and easily capable of overfitting. This means that regularization becomes more critical.

Batch normalization is applied to individual layers (optionally, to all of them) and works as follows: In each training iteration, for each layer, we first compute its activations as usual. Then, we normalize the activations of each node by subtracting its mean and dividing by its standard deviation estimating both quantities based on the statistics of the current the current minibatch.
It is precisely due to this *normalization* based on *batch* statistics that *batch normalization* derives its name.

Note that if we tried to apply BN with minibatches of size 1, we would not be able to learn anything. That is because after subtracting the means, each hidden node would take value 0! As you might guess, since we are devoting a whole

[120] https://discuss.mxnet.io/t/2357

section to BN, with large enough minibatches, the approach proves effective and stable. One takeaway here is that when applying BN, the choice of minibatch size may be even more significant than without BN.

Formally, BN transforms the activations at a given layer $\mathbf{x}$ according to the following expression:

$$\mathrm{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}}{\hat{\sigma}} + \beta \tag{9.5.1}$$

Here, $\hat{\mu}$ is the minibatch sample mean and $\hat{\sigma}$ is the minibatch sample variance. After applying BN, the resulting minibatch of activations has zero mean and unit variance. Because the choice of unit variance (vs some other magic number) is an arbitrary choice, we commonly include coordinate-wise scaling coefficients γ and offsets β. Consequently, the activation magnitudes for intermediate layers cannot diverge during training because BN actively centers and rescales them back to a given mean and size (via μ and σ). One piece of practitioner's intuition/wisdom is that BN seems to allows for more aggressive learning rates.

Formally, denoting a particular minibatch by $\mathcal{B}$, we calculate $\hat{\mu}_\mathcal{B}$ and $\hat{\sigma}_\mathcal{B}$ as follows:

$$\hat{\mu}_\mathcal{B} \leftarrow \frac{1}{|\mathcal{B}|}\sum_{\mathbf{x}\in\mathcal{B}} \mathbf{x} \text{ and } \hat{\sigma}^2_\mathcal{B} \leftarrow \frac{1}{|\mathcal{B}|}\sum_{\mathbf{x}\in\mathcal{B}} (\mathbf{x} - \mu_\mathcal{B})^2 + \epsilon \tag{9.5.2}$$

Note that we add a small constant $\epsilon > 0$ to the variance estimate to ensure that we never attempt division by zero, even in cases where the empirical variance estimate might vanish. The estimates $\hat{\mu}_\mathcal{B}$ and $\hat{\sigma}_\mathcal{B}$ counteract the scaling issue by using noisy estimates of mean and variance. You might think that this noisiness should be a problem. As it turns out, this is actually beneficial.

This turns out to be a recurring theme in deep learning. For reasons that are not yet well-characterized theoretically, various sources of noise in optimization often lead to faster training and less overfitting. While traditional machine learning theorists might buckle at this characterization, this variation appears to act as a form of regularization. In some preliminary research, (Teye et al., 2018) and (Luo et al., 2018) relate the properties of BN to Bayesian Priors and penalties respectively. In particular, this sheds some light on the puzzle of why BN works best for moderate minibatches sizes in the 50–100 range.

Fixing a trained model, you might (rightly) think that we would prefer to use the entire dataset to estimate the mean and variance. Once training is complete, why would we want the same image to be classified differently, depending on the batch in which it happens to reside? During training, such exact calculation is infeasible because the activations for all data points change every time we update our model. However, once the model is trained, we can calculate the means and variances of each layer's activations based on the entire dataset. Indeed this is standard practice for models employing batch normalization and thus MXNet's BN layers function differently in *training mode* (normalizing by minibatch statistics) and in *prediction mode* (normalizing by dataset statistics).

We are now ready to take a look at how batch normalization works in practice.

9.5.2 Batch Normalization Layers

Batch normalization implementations for fully-connected layers and convolutional layers are slightly different. We discuss both cases below. Recall that one key differences between BN and other layers is that because BN operates on a full minibatch at a time, we cannot just ignore the batch dimension as we did before when introducing other layers.

Fully-Connected Layers

When applying BN to fully-connected layers, we usually inser BN after the affine transformation and before the nonlinear activation function. Denoting the input to the layer by $\mathbf{x}$, the linear transform (with weights θ) by $f_\theta(\cdot)$, the activation function by $\phi(\cdot)$, and the BN operation with parameters β and γ by $\mathrm{BN}_{\beta,\gamma}$, we can express the computation of a BN-enabled, fully-connected layer $\mathbf{h}$ as follows:

$$\mathbf{h} = \phi(\mathrm{BN}_{\beta,\gamma}(f_\theta(\mathbf{x}))) \tag{9.5.3}$$

Recall that mean and variance are computed on the *same* minibatch $\mathcal{B}$ on which the transformation is applied. Also recall that the scaling coefficient γ and the offset β are parameters that need to be learned jointly with the more familiar parameters θ.

Convolutional Layers

Similarly, with convolutional layers, we typically apply BN after the convolution and before the nonlinear activation function. When the convolution has multiple output channels, we need to carry out batch normalization for *each* of the outputs of these channels, and each channel has its own scale and shift parameters, both of which are scalars. Assume that our minibatches contain m each and that for each channel, the output of the convolution has height p and width q. For convolutional layers, we carry out each batch normalization over the $m \cdot p \cdot q$ elements per output channel simultaneously. Thus we collect the values over all spatial locations when computing the mean and variance and consequently (within a given channel) apply the same $\hat{\mu}$ and $\hat{\sigma}$ to normalize the values at each spatial location.

Batch Normalization During Prediction

As we mentioned earlier, BN typically behaves differently in training mode and prediction mode. First, the noise in μ and σ arising from estimating each on minibatches are no longer desirable once we have trained the model. Second, we might not have the luxury of computing per-batch normalization statistics, e.g., we might need to apply our model to make one prediction at a time.

Typically, after training, we use the entire dataset to compute stable estimates of the activation statistics and then fix them at prediction time. Consequently, BN behaves differently during training and at test time Recall that dropout also exhibits this characteristic.

9.5.3 Implementation from Scratch

Below, we implement a batch normalization layer with `ndarray`s from scratch:

```
import d2l
from mxnet import autograd, gluon, np, npx, init
from mxnet.gluon import nn
npx.set_np()

def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use autograd to determine whether the current mode is training mode or
    # prediction mode
    if not autograd.is_training():
        # If it is the prediction mode, directly use the mean and variance
        # obtained from the incoming moving average
        X_hat = (X - moving_mean) / np.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # When using a fully connected layer, calculate the mean and
            # variance on the feature dimension
            mean = X.mean(axis=0)
            var = ((X - mean) ** 2).mean(axis=0)
        else:
            # When using a two-dimensional convolutional layer, calculate the
            # mean and variance on the channel dimension (axis=1). Here we
            # need to maintain the shape of X, so that the broadcast operation
            # can be carried out later
```

(continues on next page)

(continued from previous page)

```
            mean = X.mean(axis=(0, 2, 3), keepdims=True)
            var = ((X - mean) ** 2).mean(axis=(0, 2, 3), keepdims=True)
        # In training mode, the current mean and variance are used for the
        # standardization
        X_hat = (X - mean) / np.sqrt(var + eps)
        # Update the mean and variance of the moving average
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y, moving_mean, moving_var
```

We can now create a proper `BatchNorm` layer. Our layer will maintain poper parameters corresponding for scale `gamma` and shift `beta`, both of which will be updated in the course of training. Additionally, our layer will maintain a moving average of the means and variances for subsequent use during model prediction. The `num_features` parameter required by the `BatchNorm` instance is the number of outputs for a fully-connected layer and the number of output channels for a convolutional layer. The `num_dims` parameter also required by this instance is 2 for a fully-connected layer and 4 for a convolutional layer.

Putting aside the algorithmic details,
note the design pattern underlying our implementation of the layer. Typically, we define the math in a separate function, say `batch_norm`. We then integrate this functionality into a custom layer, whose code mostly addresses bookkeeping matters, such as moving data to the right device context, allocating and initializaing any required variables, keeping track of running averages (here for mean and variance), etc. This pattern enables a clean separation of math from boilerplate code. Also note that for the sake of convenience we did not worry about automagically inferring the input shape here, thus our need to specify the number of features throughout. Do not worry, the Gluon `BatchNorm` layer will care of this for us.

```
class BatchNorm(nn.Block):
    def __init__(self, num_features, num_dims, **kwargs):
        super(BatchNorm, self).__init__(**kwargs)
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # The scale parameter and the shift parameter involved in gradient
        # finding and iteration are initialized to 0 and 1 respectively
        self.gamma = self.params.get('gamma', shape=shape, init=init.One())
        self.beta = self.params.get('beta', shape=shape, init=init.Zero())
        # All the variables not involved in gradient finding and iteration are
        # initialized to 0 on the CPU
        self.moving_mean = np.zeros(shape)
        self.moving_var = np.zeros(shape)

    def forward(self, X):
        # If X is not on the CPU, copy moving_mean and moving_var to the
        # device where X is located
        if self.moving_mean.context != X.context:
            self.moving_mean = self.moving_mean.copyto(X.context)
            self.moving_var = self.moving_var.copyto(X.context)
        # Save the updated moving_mean and moving_var
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma.data(), self.beta.data(), self.moving_mean,
```

(continues on next page)

(continued from previous page)

```
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

9.5.4 Use a Batch Normalization LeNet

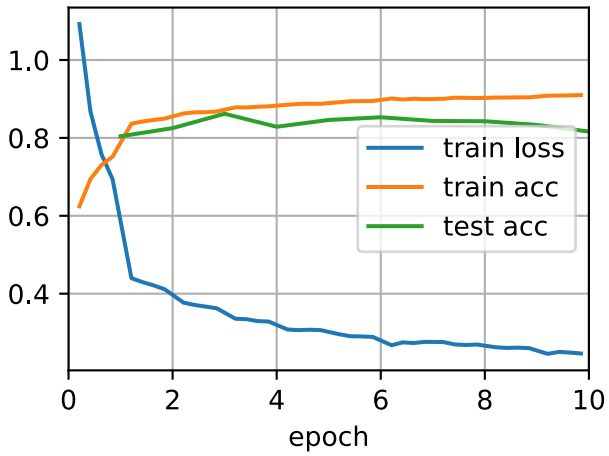
To see how to apply `BatchNorm` in context, below we apply it to a traditional LeNet model (Section 8.6). Recall that BN is typically applied after the convolutional layers and fully-connected layers but before the corresponding activation functions.

```
net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5),
        BatchNorm(6, num_dims=4),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(16, kernel_size=5),
        BatchNorm(16, num_dims=4),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120),
        BatchNorm(120, num_dims=2),
        nn.Activation('sigmoid'),
        nn.Dense(84),
        BatchNorm(84, num_dims=2),
        nn.Activation('sigmoid'),
        nn.Dense(10))
```

As before, we will train our network on the Fashion-MNIST dataset. This code is virtually identical to that when we first trained LeNet (Section 8.6). The main difference is the considerably larger learning rate.

```
lr, num_epochs, batch_size = 1.0, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.248, train acc 0.909, test acc 0.816
23242.0 exampes/sec on gpu(0)
```



Let us have a look at the scale parameter `gamma` and the shift parameter `beta` learned from the first batch normalization layer.

```
net[1].gamma.data().reshape(-1,), net[1].beta.data().reshape(-1,)
```

```
(array([0.8818444, 1.2196997, 2.9328399, 1.8038002, 0.752473 , 1.8539152],␣
↪ctx=gpu(0)),
 array([ 0.82514095,  0.09633566, -3.0784628 , -0.27704707, -0.8937671 ,
        -0.22957063], ctx=gpu(0)))
```

9.5.5 Concise Implementation

Compared with the `BatchNorm` class, which we just defined ourselves, the `BatchNorm` class defined by the `nn` model in Gluon is easier to use. In Gluon, we do not have to worry about `num_features` or `num_dims`. Instead, these parameter values will be inferred automatically via delayed initialization. Otherwise, the code looks virtually identical to the application our implementation above.

```
net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(16, kernel_size=5),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.Dense(84),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.Dense(10))
```

Below, we use the same hyper-parameters to train out model. Note that as usual, the Gluon variant runs much faster because its code has been compiled to C++/CUDA while our custom implementation must be interpreted by Python.

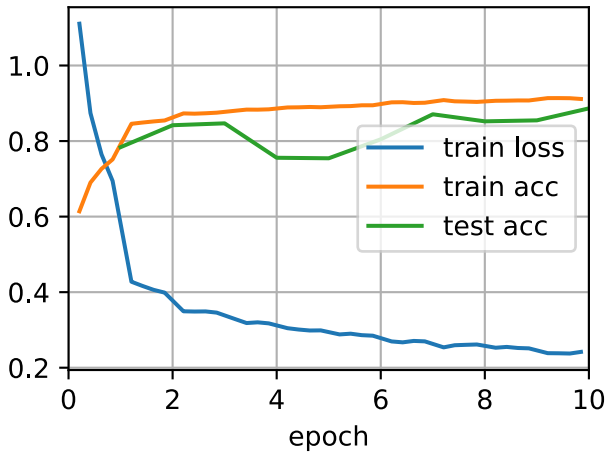
```
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.243, train acc 0.911, test acc 0.886
45057.0 exampes/sec on gpu(0)
```

9.5.6 Controversy

Intuitively, batch normalization is thought to make the optimization landscape smoother. However, we must be careful to distinguish between speculative intuitions and true explanations for the phenomena that we observe when training deep models. Recall that we do not even know why simpler deep neural networks (MLPs and conventional CNNs) generalize well in the first place. Even with dropout and L2 regularization, they remain so flexible that their ability to generalize to unseen data cannot be explained via conventional learning-theoretic generalization guarantees.

In the original paper proposing batch normalization, the authors, in addition to introducing a powerful and useful tool, offered an explanation for why it works: by reducing *internal covariate shift*. Presumably by *internal covariate shift* the



authors meant something like the intuition expressed above—the notion that the distribution of activations changes over the course of training. However there were two problems with this explanation: (1) This drift is very different from *covariate shift*, rendering the name a misnomer. (2) The explanation offers an under-specified intuition but leaves the question of *why precisely this technique works* an open question wanting for a rigorous explanation. Throughout this book, we aim to convey the intuitions that practitioners use to guide their development of deep neural networks. However, we believe that it is important to separate these guiding intuitions from established sceintific fact. Eventually, when you master this material and start writing your own research papers you will want to be clear to delineate between technical claims and hunches.

Following the success of batch normalization, its explanation in terms of *internal covariate shift* has repeatedly surfaced in debates in the technical literature and broader discourse about how to present machine learning research. In a memorable speech given while accepting a Test of Time Award at the 2017 NeurIPS conference, Ali Rahimi used *internal covariate shift* as a focal point in an argument likening the modern practice of deep learning to alchemy. Subsequently, the example was revisited in detail in a position paper outlining troubling trends in machine learning (Lipton & Steinhardt, 2018).
In the technical literature other authors ((Santurkar et al., 2018)) have proposed alternative explanations for the success of BN, some claiming that BN's success comes despite exhibiting behavior that is in some ways opposite to those claimed in the original paper.

We note that the *internal covariate shift* is no more worthy of criticism than any of thousands of similarly vague claims made every year in the technical ML literature. Likely, its resonance as a focal point of these debates owes to its broad recognizability to the target audience. Batch normalization has proven an indispensable method, applied in nearly all deployed image classifiers, earning the paper that introduced the technique tens of thousands of citations.

9.5.7 Summary

- During model training, batch normalization continuously adjusts the intermediate output of the neural network by utilizing the mean and standard deviation of the minibatch, so that the values of the intermediate output in each layer throughout the neural network are more stable.
- The batch normalization methods for fully connected layers and convolutional layers are slightly different.
- Like a dropout layer, batch normalization layers have different computation results in training mode and prediction mode.

- Batch Normalization has many beneficial side effects, primarily that of regularization. On the other hand, the original motivation of reducing covariate shift seems not to be a valid explanation.

9.5.8 Exercises

1. Can we remove the fully connected affine transformation before the batch normalization or the bias parameter in convolution computation?
   - Find an equivalent transformation that applies prior to the fully connected layer.
   - Is this reformulation effective. Why (not)?
2. Compare the learning rates for LeNet with and without batch normalization.
   - Plot the decrease in training and test error.
   - What about the region of convergence? How large can you make the learning rate?
3. Do we need Batch Normalization in every layer? Experiment with it?
4. Can you replace Dropout by Batch Normalization? How does the behavior change?
5. Fix the coefficients `beta` and `gamma` (add the parameter `grad_req='null'` at the time of construction to avoid calculating the gradient), and observe and analyze the results.
6. Review the Gluon documentation for `BatchNorm` to see the other applications for Batch Normalization.
7. Research ideas - think of other normalization transforms that you can apply? Can you apply the probability integral transform? How about a full rank covariance estimate?

9.5.9 Scan the QR Code to Discuss[121]

9.6 Residual Networks (ResNet)

As we design increasingly deeper networks it becomes imperative to understand how adding layers can increase the complexity and expressiveness of the network. Even more important is the ability to design networks where adding layers makes networks strictly more expressive rather than just different. To make some progress we need a bit of theory.

9.6.1 Function Classes

Consider $\mathcal{F}$, the class of functions that a specific network architecture (together with learning rates and other hyperparameter settings) can reach. That is, for all $f \in \mathcal{F}$ there exists some set of parameters W that can be obtained through training on a suitable dataset. Let us assume that f^* is the function that we really would like to find. If it is in $\mathcal{F}$, we are in good shape but typically we will not be quite so lucky. Instead, we will try to find some $f^*_{\mathcal{F}}$ which is our best bet within $\mathcal{F}$. For instance, we might try finding it by solving the following optimization problem:

$$f^*_{\mathcal{F}} := \operatorname*{argmin}_f L(X, Y, f) \text{ subject to } f \in \mathcal{F} \tag{9.6.1}$$

[121] https://discuss.mxnet.io/t/2358

It is only reasonable to assume that if we design a different and more powerful architecture $\mathcal{F}'$ we should arrive at a better outcome. In other words, we would expect that $f^*_{\mathcal{F}'}$ is 'better' than $f^*_{\mathcal{F}}$. However, if $\mathcal{F} \not\subseteq \mathcal{F}'$ there is no guarantee that this should even happen. In fact, $f^*_{\mathcal{F}'}$ might well be worse. This is a situation that we often encounter in practice - adding layers does not only make the network more expressive, it also changes it in sometimes not quite so predictable ways. The picture below illustrates this in slightly abstract terms.

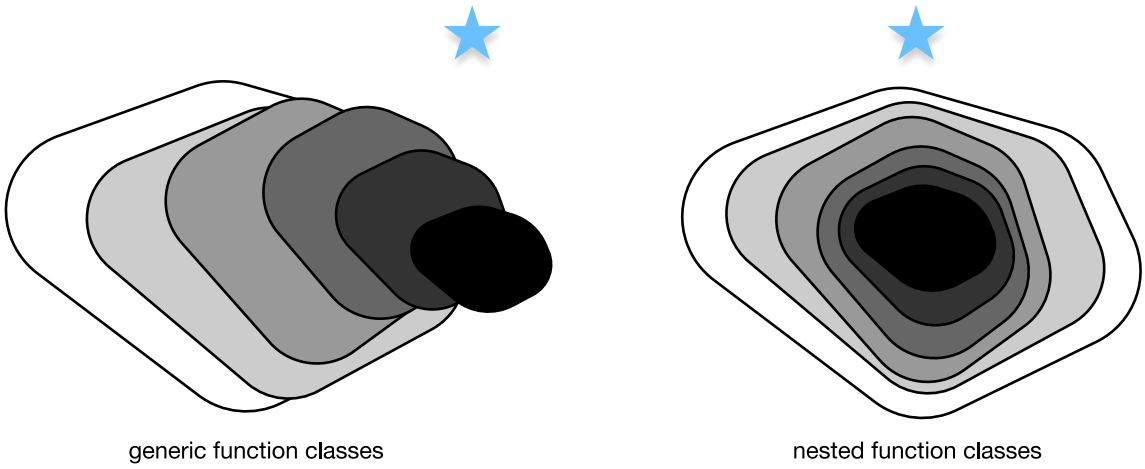


Fig. 9.6.1: Left: non-nested function classes. The distance may in fact increase as the complexity increases. Right: with nested function classes this does not happen.

Only if larger function classes contain the smaller ones are we guaranteed that increasing them strictly increases the expressive power of the network. This is the question that He et al, 2016 considered when working on very deep computer vision models. At the heart of ResNet is the idea that every additional layer should contain the identity function as one of its elements. This means that if we can train the newly-added layer into an identity mapping $f(\mathbf{x}) = \mathbf{x}$, the new model will be as effective as the original model. As the new model may get a better solution to fit the training dataset, the added layer might make it easier to reduce training errors. Even better, the identity function rather than the null $f(\mathbf{x}) = 0$ should be the the simplest function within a layer.

These considerations are rather profound but they led to a surprisingly simple solution, a residual block. With it, (He et al., 2016a) won the ImageNet Visual Recognition Challenge in 2015. The design had a profound influence on how to build deep neural networks.

9.6.2 Residual Blocks

Let us focus on a local neural network, as depicted below. Denote the input by $\mathbf{x}$. We assume that the ideal mapping we want to obtain by learning is $f(\mathbf{x})$, to be used as the input to the activation function. The portion within the dotted-line box in the left image must directly fit the mapping $f(\mathbf{x})$. This can be tricky if we do not need that particular layer and we would much rather retain the input $\mathbf{x}$. The portion within the dotted-line box in the right image now only needs to parametrize the *deviation* from the identity, since we return $\mathbf{x} + f(\mathbf{x})$. In practice, the residual mapping is often easier to optimize. We only need to set $f(\mathbf{x}) = 0$. The right image in the figure below illustrates the basic Residual Block of ResNet. Similar architectures were later proposed for sequence models which we will study later.

ResNet follows VGG's full 3×3 convolutional layer design. The residual block has two 3×3 convolutional layers with the same number of output channels. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function. This kind of design requires that the output of the two convolutional layers be of the same shape as the input, so that they can be added together. If we want to change the number of channels or the the stride, we need to introduce an additional 1×1 convolutional layer to transform the input into the desired shape for the addition operation. Let us have a look at the code below.

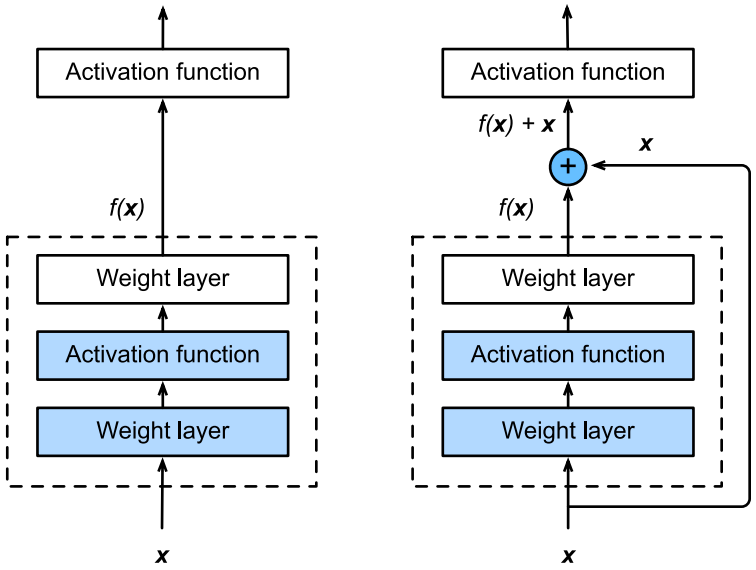


Fig. 9.6.2: The difference between a regular block (left) and a residual block (right). In the latter case, we can short-circuit the convolutions.

```
import d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
npx.set_np()

# Saved in the d2l package for later use
class Residual(nn.Block):
    def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                               strides=strides)
        self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                   strides=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def forward(self, X):
        Y = npx.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return npx.relu(Y + X)
```

This code generates two types of networks: one where we add the input to the output before applying the ReLU nonlinearity, and whenever `use_1x1conv=True`, one where we adjust channels and resolution by means of a 1×1 convolution before adding. The diagram below illustrates this:

Now let us look at a situation where the input and output are of the same shape.

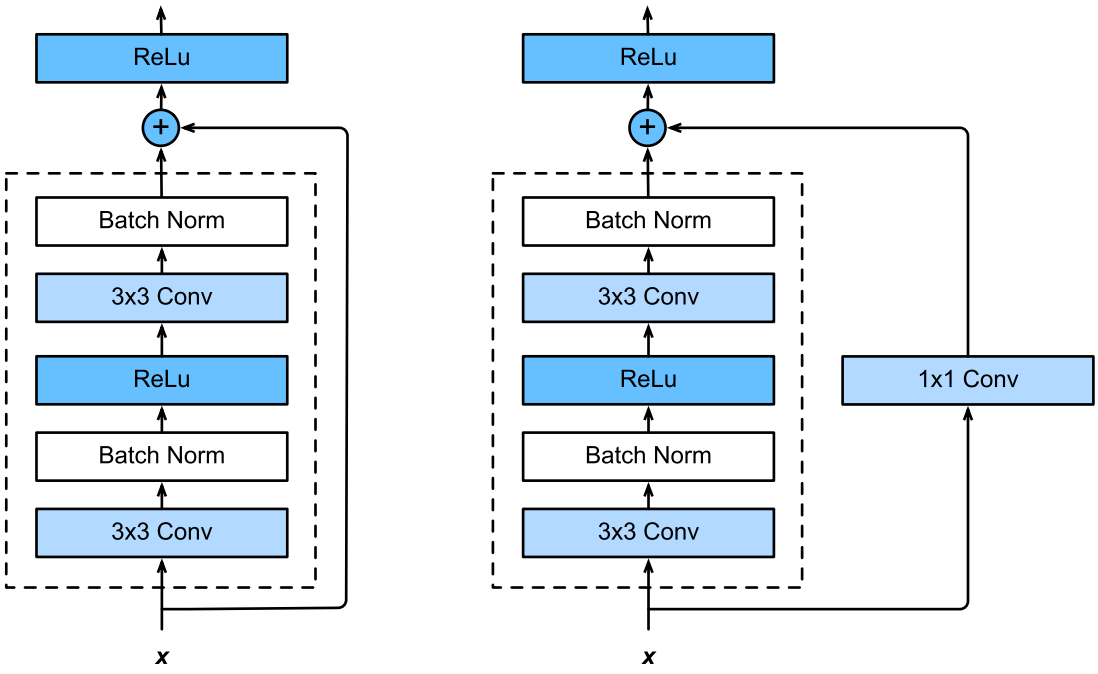


Fig. 9.6.3: Left: regular ResNet block; Right: ResNet block with 1x1 convolution

```
blk = Residual(3)
blk.initialize()
X = np.random.uniform(size=(4, 3, 6, 6))
blk(X).shape
```

```
(4, 3, 6, 6)
```

We also have the option to halve the output height and width while increasing the number of output channels.

```
blk = Residual(6, use_1x1conv=True, strides=2)
blk.initialize()
blk(X).shape
```

```
(4, 6, 3, 3)
```

9.6.3 ResNet Model

The first two layers of ResNet are the same as those of the GoogLeNet we described before: the 7×7 convolutional layer with 64 output channels and a stride of 2 is followed by the 3×3 maximum pooling layer with a stride of 2. The difference is the batch normalization layer added after each convolutional layer in ResNet.

```
net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
        nn.BatchNorm(), nn.Activation('relu'),
        nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

GoogLeNet uses four blocks made up of Inception blocks. However, ResNet uses four modules made up of residual blocks, each of which uses several residual blocks with the same number of output channels. The number of channels in the first module is the same as the number of input channels. Since a maximum pooling layer with a stride of 2 has already been used, it is not necessary to reduce the height and width. In the first residual block for each of the subsequent modules, the number of channels is doubled compared with that of the previous module, and the height and width are halved.

Now, we implement this module. Note that special processing has been performed on the first module.

```
def resnet_block(num_channels, num_residuals, first_block=False):
    blk = nn.Sequential()
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.add(Residual(num_channels))
    return blk
```

Then, we add all the residual blocks to ResNet. Here, two residual blocks are used for each module.

```
net.add(resnet_block(64, 2, first_block=True),
        resnet_block(128, 2),
        resnet_block(256, 2),
        resnet_block(512, 2))
```

Finally, just like GoogLeNet, we add a global average pooling layer, followed by the fully connected layer output.

```
net.add(nn.GlobalAvgPool2D(), nn.Dense(10))
```

There are 4 convolutional layers in each module (excluding the 1×1 convolutional layer). Together with the first convolutional layer and the final fully connected layer, there are 18 layers in total. Therefore, this model is commonly known as ResNet-18. By configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as the deeper 152-layer ResNet-152. Although the main architecture of ResNet is similar to that of GoogLeNet, ResNet's structure is simpler and easier to modify. All these factors have resulted in the rapid and widespread use of ResNet. Below is a diagram of the full ResNet-18.

Before training ResNet, let us observe how the input shape changes between different modules in ResNet. As in all previous architectures, the resolution decreases while the number of channels increases up until the point where a global average pooling layer aggregates all features.

```
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)
```

```
conv5 output shape:  (1, 64, 112, 112)
batchnorm4 output shape:     (1, 64, 112, 112)
relu0 output shape:  (1, 64, 112, 112)
pool0 output shape:  (1, 64, 56, 56)
sequential1 output shape:    (1, 64, 56, 56)
sequential2 output shape:    (1, 128, 28, 28)
sequential3 output shape:    (1, 256, 14, 14)
sequential4 output shape:    (1, 512, 7, 7)
pool1 output shape:  (1, 512, 1, 1)
dense0 output shape:         (1, 10)
```

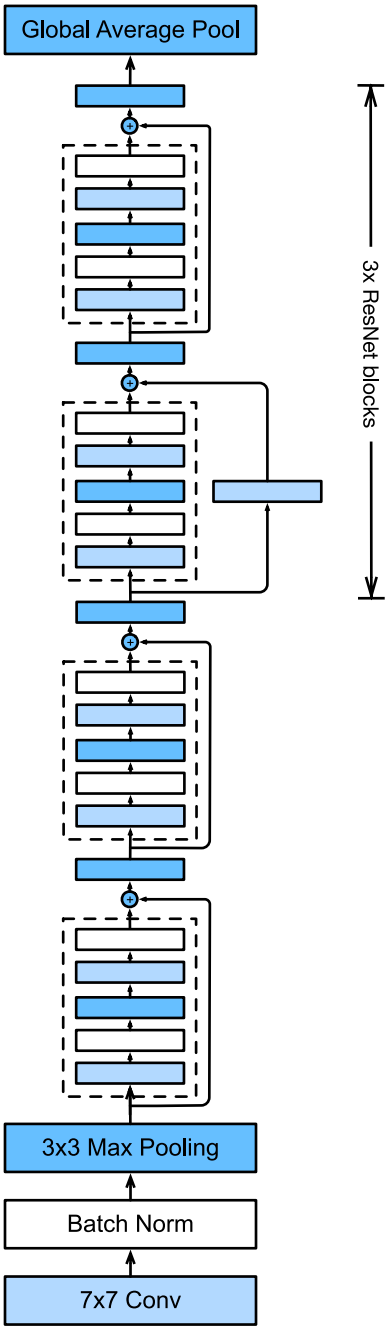


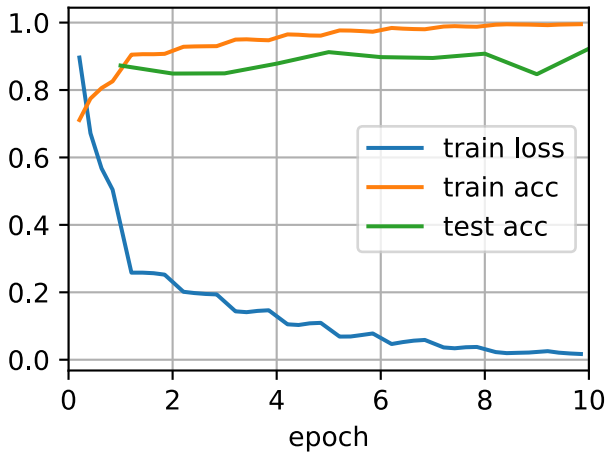
Fig. 9.6.4: ResNet 18

9.6.4 Data Acquisition and Training

We train ResNet on the Fashion-MNIST dataset, just like before. The only thing that has changed is the learning rate that decreased again, due to the more complex architecture.

```
lr, num_epochs, batch_size = 0.05, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.017, train acc 0.995, test acc 0.923
4939.9 exampes/sec on gpu(0)
```



9.6.5 Summary

- Residual blocks allow for a parametrization relative to the identity function $f(\mathbf{x}) = \mathbf{x}$.
- Adding residual blocks increases the function complexity in a well-defined manner.
- We can train an effective deep neural network by having residual blocks pass through cross-layer data channels.
- ResNet had a major influence on the design of subsequent deep neural networks, both for convolutional and sequential nature.

9.6.6 Exercises

1. Refer to Table 1 in the (He et al., 2016a) to implement different variants.
2. For deeper networks, ResNet introduces a "bottleneck" architecture to reduce model complexity. Try to implement it.
3. In subsequent versions of ResNet, the author changed the "convolution, batch normalization, and activation" architecture to the "batch normalization, activation, and convolution" architecture. Make this improvement yourself. See Figure 1 in (He et al., 2016b) for details.
4. Prove that if $\mathbf{x}$ is generated by a ReLU, the ResNet block does indeed include the identity function.
5. Why cannot we just increase the complexity of functions without bound, even if the function classes are nested?

9.6.7 Scan the QR Code to Discuss[122]

9.7 Densely Connected Networks (DenseNet)

ResNet significantly changed the view of how to parametrize the functions in deep networks. DenseNet is to some extent the logical extension of this. To understand how to arrive at it, let us take a small detour to theory. Recall the Taylor expansion for functions. For scalars it can be written as

$$f(x) = f(0) + f'(x)x + \frac{1}{2}f''(x)x^2 + \frac{1}{6}f'''(x)x^3 + o(x^3) \tag{9.7.1}$$

9.7.1 Function Decomposition

The key point is that it decomposes the function into increasingly higher order terms. In a similar vein, ResNet decomposes functions into

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}) \tag{9.7.2}$$

That is, ResNet decomposes f into a simple linear term and a more complex nonlinear one. What if we want to go beyond two terms? A solution was proposed by (Huang et al., 2017) in the form of DenseNet, an architecture that reported record performance on the ImageNet dataset.

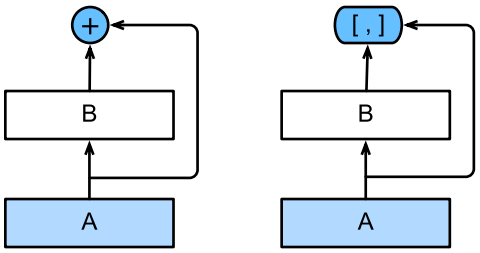


Fig. 9.7.1: The main difference between ResNet (left) and DenseNet (right) in cross-layer connections: use of addition and use of concatenation.

The key difference between ResNet and DenseNet is that in the latter case outputs are *concatenated* rather than added. As a result we perform a mapping from $\mathbf{x}$ to its values after applying an increasingly complex sequence of functions.

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), f_3(\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), \ldots] \tag{9.7.3}$$

In the end, all these functions are combined in an MLP to reduce the number of features again. In terms of implementation this is quite simple - rather than adding terms, we concatenate them. The name DenseNet arises from the fact that the dependency graph between variables becomes quite dense. The last layer of such a chain is densely connected to all previous layers. The main components that compose a DenseNet are dense blocks and transition layers. The former defines how the inputs and outputs are concatenated, while the latter controls the number of channels so that it is not too large.

[122] https://discuss.mxnet.io/t/2359

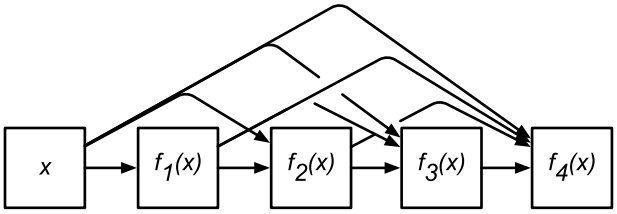


Fig. 9.7.2: Dense connections in DenseNet

9.7.2 Dense Blocks

DenseNet uses the modified "batch normalization, activation, and convolution" architecture of ResNet (see the exercise in Section 9.6). First, we implement this architecture in the `conv_block` function.

```
import d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
npx.set_np()

def conv_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(),
            nn.Activation('relu'),
            nn.Conv2D(num_channels, kernel_size=3, padding=1))
    return blk
```

A dense block consists of multiple `conv_block` units, each using the same number of output channels. In the forward computation, however, we concatenate the input and output of each block on the channel dimension.

```
class DenseBlock(nn.Block):
    def __init__(self, num_convs, num_channels, **kwargs):
        super(DenseBlock, self).__init__(**kwargs)
        self.net = nn.Sequential()
        for _ in range(num_convs):
            self.net.add(conv_block(num_channels))

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # Concatenate the input and output of each block on the channel
            # dimension
            X = np.concatenate((X, Y), axis=1)
        return X
```

In the following example, we define a convolution block with two blocks of 10 output channels. When using an input with 3 channels, we will get an output with the $3 + 2 \times 10 = 23$ channels. The number of convolution block channels controls the increase in the number of output channels relative to the number of input channels. This is also referred to as the growth rate.

```
blk = DenseBlock(2, 10)
blk.initialize()
X = np.random.uniform(size=(4, 3, 8, 8))
Y = blk(X)
```

(continues on next page)

(continued from previous page)

```
Y.shape
```

```
(4, 23, 8, 8)
```

9.7.3 Transition Layers

Since each dense block will increase the number of channels, adding too many of them will lead to an excessively complex model. A transition layer is used to control the complexity of the model. It reduces the number of channels by using the 1×1 convolutional layer and halves the height and width of the average pooling layer with a stride of 2, further reducing the complexity of the model.

```
def transition_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(), nn.Activation('relu'),
            nn.Conv2D(num_channels, kernel_size=1),
            nn.AvgPool2D(pool_size=2, strides=2))
    return blk
```

Apply a transition layer with 10 channels to the output of the dense block in the previous example. This reduces the number of output channels to 10, and halves the height and width.

```
blk = transition_block(10)
blk.initialize()
blk(Y).shape
```

```
(4, 10, 4, 4)
```

9.7.4 DenseNet Model

Next, we will construct a DenseNet model. DenseNet first uses the same single convolutional layer and maximum pooling layer as ResNet.

```
net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
        nn.BatchNorm(), nn.Activation('relu'),
        nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

Then, similar to the four residual blocks that ResNet uses, DenseNet uses four dense blocks. Similar to ResNet, we can set the number of convolutional layers used in each dense block. Here, we set it to 4, consistent with the ResNet-18 in the previous section. Furthermore, we set the number of channels (i.e., growth rate) for the convolutional layers in the dense block to 32, so 128 channels will be added to each dense block.

In ResNet, the height and width are reduced between each module by a residual block with a stride of 2. Here, we use the transition layer to halve the height and width and halve the number of channels.

```
# Num_channels: the current number of channels
num_channels, growth_rate = 64, 32
num_convs_in_dense_blocks = [4, 4, 4, 4]
```

(continues on next page)

(continued from previous page)

```
for i, num_convs in enumerate(num_convs_in_dense_blocks):
    net.add(DenseBlock(num_convs, growth_rate))
    # This is the number of output channels in the previous dense block
    num_channels += num_convs * growth_rate
    # A transition layer that haves the number of channels is added between
    # the dense blocks
    if i != len(num_convs_in_dense_blocks) - 1:
        num_channels //= 2
        net.add(transition_block(num_channels))
```

Similar to ResNet, a global pooling layer and fully connected layer are connected at the end to produce the output.

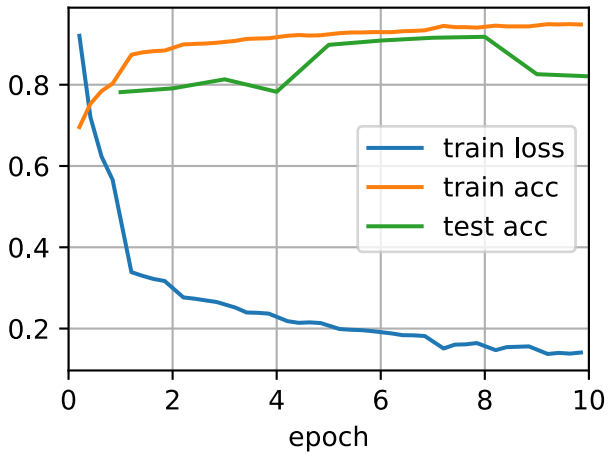
```
net.add(nn.BatchNorm(),
        nn.Activation('relu'),
        nn.GlobalAvgPool2D(),
        nn.Dense(10))
```

9.7.5 Data Acquisition and Training

Since we are using a deeper network here, in this section, we will reduce the input height and width from 224 to 96 to simplify the computation.

```
lr, num_epochs, batch_size = 0.1, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.141, train acc 0.949, test acc 0.821
5669.1 exampes/sec on gpu(0)
```



9.7.6 Summary

- In terms of cross-layer connections, unlike ResNet, where inputs and outputs are added together, DenseNet concatenates inputs and outputs on the channel dimension.

- The main units that compose DenseNet are dense blocks and transition layers.
- We need to keep the dimensionality under control when composing the network by adding transition layers that shrink the number of channels again.

9.7.7 Exercises

1. Why do we use average pooling rather than max-pooling in the transition layer?
2. One of the advantages mentioned in the DenseNet paper is that its model parameters are smaller than those of ResNet. Why is this the case?
3. One problem for which DenseNet has been criticized is its high memory consumption.
   - Is this really the case? Try to change the input shape to 224×224 to see the actual (GPU) memory consumption.
   - Can you think of an alternative means of reducing the memory consumption? How would you need to change the framework?
4. Implement the various DenseNet versions presented in Table 1 of (Huang et al., 2017).
5. Why do we not need to concatenate terms if we are just interested in $\mathbf{x}$ and $f(\mathbf{x})$ for ResNet? Why do we need this for more than two layers in DenseNet?
6. Design a DenseNet for fully connected networks and apply it to the Housing Price prediction task.

9.7.8 Scan the QR Code to Discuss[123]

[123] https://discuss.mxnet.io/t/2360

CHAPTER

TEN

RECURRENT NEURAL NETWORKS

So far we encountered two types of data: generic vectors and images. For the latter we designed specialized layers to take advantage of the regularity properties in them. In other words, if we were to permute the pixels in an image, it would be much more difficult to reason about its content of something that would look much like the background of a test pattern in the times of Analog TV.

Most importantly, so far we tacitly assumed that our data is generated i.i.d., i.e., independently and identically distributed, all drawn from some distribution. Unfortunately, this is not true for most data. For instance, the words in this paragraph are written in sequence, and it would be quite difficult to decipher its meaning if they were permuted randomly. Likewise, image frames in a video, the audio signal in a conversation, or the browsing behavior on a website, all follow sequential order. It is thus only reasonable to assume that specialized models for such data will do better at describing it and at solving estimation problems.

Another issue arises from the fact that we might not only receive a sequence as an input but rather might be expected to continue the sequence. For instance, the task could be to continue the series 2, 4, 6, 8, 10, … This is quite common in time series analysis, to predict the stock market, the fever curve of a patient or the acceleration needed for a race car. Again we want to have models that can handle such data.

In short, while convolutional neural networks can efficiently process spatial information, recurrent neural networks are designed to better handle sequential information. These networks introduce state variables to store past information and, together with the current input, determine the current output.

Many of the examples for using recurrent networks are based on text data. Hence, we will emphasize language models in this chapter. After a more formal review of sequence data we discuss basic concepts of a language model and use this discussion as the inspiration for the design of recurrent neural networks. Next, we describe the gradient calculation method in recurrent neural networks to explore problems that may be encountered in recurrent neural network training. For some of these problems, we can use gated recurrent neural networks, such as LSTMs (Section 10.9) and GRUs (Section 10.8), described later in this chapter.

10.1 Sequence Models

Imagine that you are watching movies on Netflix. As a good Netflix user you decide to rate each of the movies religiously. After all, a good movie is a good movie, and you want to watch more of them, right? As it turns out, things are not quite so simple. People's opinions on movies can change quite significantly over time. In fact, psychologists even have names for some of the effects:

- There is anchoring[124], based on someone else's opinion. For instance after the Oscar awards, ratings for the corresponding movie go up, even though it is still the same movie. This effect persists for a few months until the award is forgotten. (Wu et al., 2017) showed that the effect lifts rating by over half a point.

[124] https://en.wikipedia.org/wiki/Anchoring

- There is the Hedonic adaptation[125], where humans quickly adapt to accept an improved (or a bad) situation as the new normal. For instance, after watching many good movies, the expectations that the next movie be equally good or better are high, and even an average movie might be considered a bad movie after many great ones.
- There is seasonality. Very few viewers like to watch a Santa Claus movie in August.
- In some cases movies become unpopular due to the misbehaviors of directors or actors in the production.
- Some movies become cult movies, because they were almost comically bad. *Plan 9 from Outer Space* and *Troll 2* achieved a high degree of notoriety for this reason.

In short, ratings are anything but stationary. Using temporal dynamics helped (Koren, 2009) to recommend movies more accurately. But it is not just about movies.

- Many users have highly particular behavior when it comes to the time when they open apps. For instance, social media apps are much more popular after school with students. Stock market trading apps are more commonly used when the markets are open.
- It is much harder to predict tomorrow's stock prices than to fill in the blanks for a stock price we missed yesterday, even though both are just a matter of estimating one number. After all, hindsight is so much easier than foresight. In statistics the former is called *prediction* whereas the latter is called *filtering*.
- Music, speech, text, movies, steps, etc. are all sequential in nature. If we were to permute them they would make little sense. The headline *dog bites man* is much less surprising than *man bites dog*, even though the words are identical.
- Earthquakes are strongly correlated, i.e., after a massive earthquake there are very likely several smaller aftershocks, much more so than without the strong quake. In fact, earthquakes are spatiotemporally correlated, i.e., the aftershocks typically occur within a short time span and in close proximity.
- Humans interact with each other in a sequential nature, as can be seen in Twitter fights, dance patterns and debates.

10.1.1 Statistical Tools

In short, we need statistical tools and new deep networks architectures to deal with sequence data. To keep things simple, we use the stock price as an example.

Let us denote the prices by $x_t \geq 0$, i.e., at time $t \in \mathbb{N}$ we observe some price x_t. For a trader to do well in the stock market on day t he should want to predict x_t via

$$x_t \sim p(x_t \mid x_{t-1}, \ldots, x_1). \tag{10.1.1}$$

Autoregressive Models

In order to achieve this, our trader could use a regressor such as the one we trained in Section 5.3. There is just a major problem - the number of inputs, $x_{t-1}, \ldots, x_1$ varies, depending on t. That is, the number increases with the amount of data that we encounter, and we will need an approximation to make this computationally tractable. Much of what follows in this chapter will revolve around how to estimate $p(x_t \mid x_{t-1}, \ldots, x_1)$ efficiently. In a nutshell it boils down to two strategies:

1. Assume that the potentially rather long sequence $x_{t-1}, \ldots, x_1$ is not really necessary. In this case we might content ourselves with some timespan τ and only use $x_{t-1}, \ldots, x_{t-\tau}$ observations. The immediate benefit is that now the number of arguments is always the same, at least for $t > \tau$. This allows us to train a deep network as indicated above. Such models will be called *autoregressive* models, as they quite literally perform regression on themselves.

[125] https://en.wikipedia.org/wiki/Hedonic_treadmill

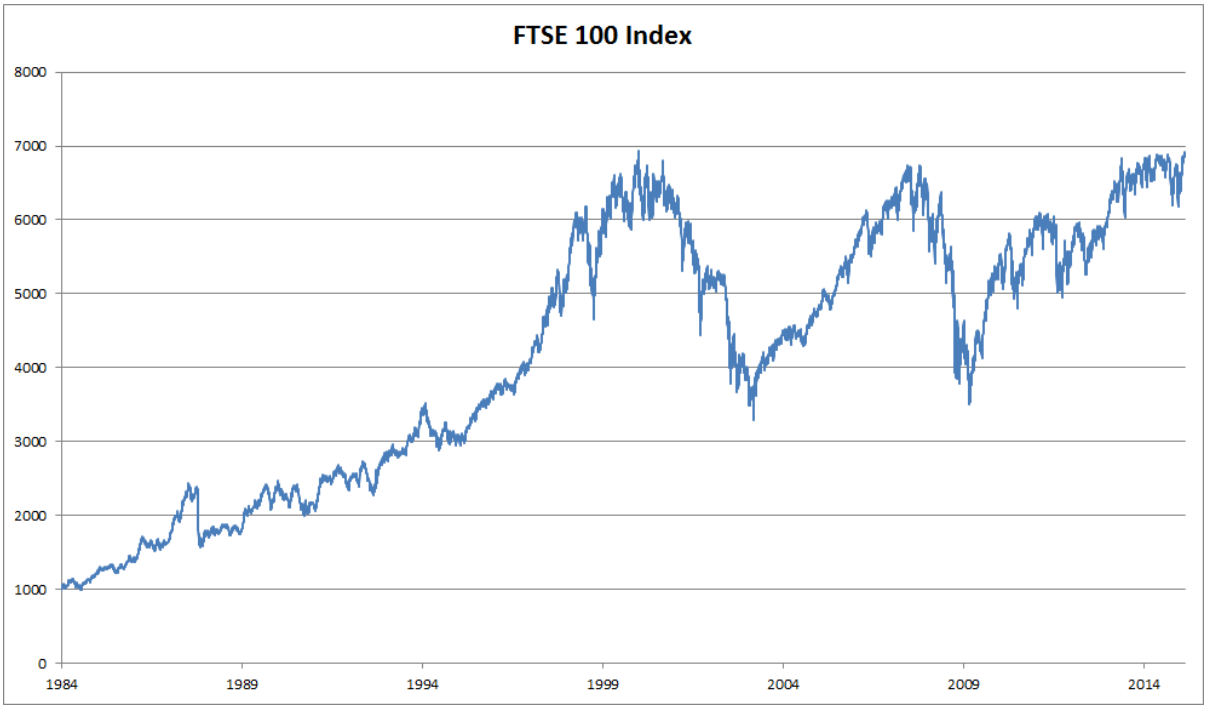


Fig. 10.1.1: FTSE 100 index over 30 years

2. Another strategy, shown in Fig. 10.1.2, is to try and keep some summary h_t of the past observations, at the same time update h_t in addition to the prediction $\hat{x}_t$. This leads to models that estimate x_t with $\hat{x}_t = p(x_t \mid x_{t-1}, h_t)$ and moreover updates of the form $h_t = g(h_{t-1}, x_{t-1})$. Since h_t is never observed, these models are also called *latent autoregressive models*. LSTMs and GRUs are examples of this.

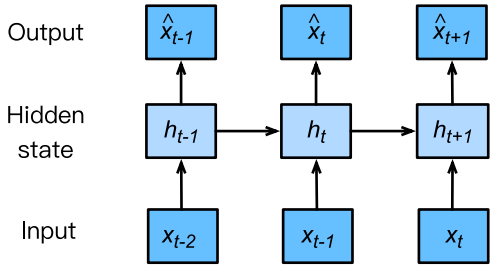


Fig. 10.1.2: A latent autoregressive model.

Both cases raise the obvious question how to generate training data. One typically uses historical observations to predict the next observation given the ones up to right now. Obviously we do not expect time to stand still. However, a common assumption is that while the specific values of x_t might change, at least the dynamics of the time series itself will not. This is reasonable, since novel dynamics are just that, novel and thus not predictable using data we have so far. Statisticians call dynamics that do not change *stationary*. Regardless of what we do, we will thus get an estimate of the entire time series via

$$p(x_1, \ldots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_{t-1}, \ldots, x_1). \tag{10.1.2}$$

Note that the above considerations still hold if we deal with discrete objects, such as words, rather than numbers. The only difference is that in such a situation we need to use a classifier rather than a regressor to estimate $p(x_t \mid x_{t-1}, \ldots, x_1)$.

Markov Model

Recall the approximation that in an autoregressive model we use only $(x_{t-1}, \ldots, x_{t-\tau})$ instead of $(x_{t-1}, \ldots, x_1)$ to estimate x_t. Whenever this approximation is accurate we say that the sequence satisfies a Markov condition. In particular, if $\tau = 1$, we have a *first order* Markov model and $p(x)$ is given by

$$p(x_1, \ldots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_{t-1}). \tag{10.1.3}$$

Such models are particularly nice whenever x_t assumes only discrete values, since in this case dynamic programming can be used to compute values along the chain exactly. For instance, we can compute $p(x_{t+1} \mid x_{t-1})$ efficiently using the fact that we only need to take into account a very short history of past observations:

$$p(x_{t+1} \mid x_{t-1}) = \sum_{x_t} p(x_{t+1} \mid x_t) p(x_t \mid x_{t-1}). \tag{10.1.4}$$

Going into details of dynamic programming is beyond the scope of this section. Control[126] and reinforcement learning[127] algorithms use such tools extensively.

Causality

In principle, there is nothing wrong with unfolding $p(x_1, \ldots, x_T)$ in reverse order. After all, by conditioning we can always write it via

$$p(x_1, \ldots, x_T) = \prod_{t=T}^{1} p(x_t \mid x_{t+1}, \ldots, x_T). \tag{10.1.5}$$

In fact, if we have a Markov model, we can obtain a reverse conditional probability distribution, too. In many cases, however, there exists a natural direction for the data, namely going forward in time. It is clear that future events cannot influence the past. Hence, if we change x_t, we may be able to influence what happens for x_{t+1} going forward but not the converse. That is, if we change x_t, the distribution over past events will not change. Consequently, it ought to be easier to explain $p(x_{t+1} \mid x_t)$ rather than $p(x_t \mid x_{t+1})$. For instance, Hoyer et al., 2008[128] show that in some cases we can find $x_{t+1} = f(x_t) + \epsilon$ for some additive noise, whereas the converse is not true. This is great news, since it is typically the forward direction that we are interested in estimating. For more on this topic see e.g., the book by Peters, Janzing and Schölkopf, 2015[129]. We are barely scratching the surface of it.

10.1.2 Toy Example

After so much theory, let us try this out in practice. Since much of the modeling is identical to when we built regression estimators in Gluon, we will not delve into much detail regarding the choice of architecture besides the fact that we will use several layers of a fully connected network. Let us begin by generating some data. To keep things simple we generate our 'time series' by using a sine function with some additive noise.

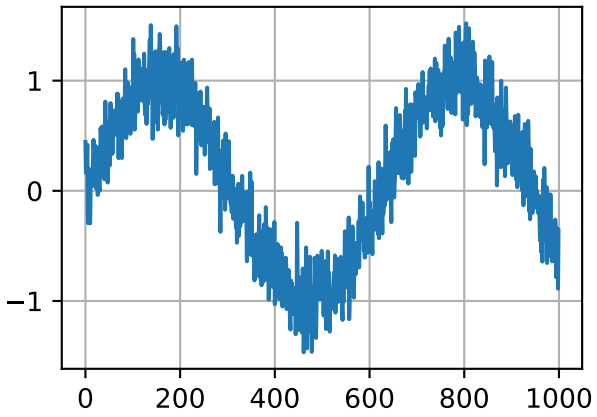
```
%matplotlib inline
import d2l
from mxnet import autograd, np, npx, gluon, init
from mxnet.gluon import nn
npx.set_np()
```

(continues on next page)

[126] https://en.wikipedia.org/wiki/Control_theory
[127] https://en.wikipedia.org/wiki/Reinforcement_learning
[128] https://papers.nips.cc/paper/3548-nonlinear-causal-discovery-with-additive-noise-models
[129] https://mitpress.mit.edu/books/elements-causal-inference

(continued from previous page)

```
T = 1000  # Generate a total of 1000 points
time = np.arange(0, T)
x = np.sin(0.01 * time) + 0.2 * np.random.normal(size=T)
d2l.plot(time, [x])
```



Next we need to turn this 'time series' into data the network can train on. Based on the embedding dimension τ we map the data into pairs $y_t = x_t$ and $\mathbf{z}_t = (x_{t-1}, \ldots, x_{t-\tau})$. The astute reader might have noticed that this gives us τ fewer datapoints, since we do not have sufficient history for the first τ of them. A simple fix, in particular if the time series is long is to discard those few terms. Alternatively we could pad the time series with zeros. The code below is essentially identical to the training code in previous sections. We kept the architecture fairly simple. A few layers of a fully connected network, ReLU activation and ℓ_2 loss.

```
tau = 4
features = np.zeros((T-tau, tau))
for i in range(tau):
    features[:, i] = x[i: T-tau+i]
labels = x[tau:]

batch_size, n_train = 16, 600
train_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                            batch_size, is_train=True)
test_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                           batch_size, is_train=False)

# Vanilla MLP architecture
def get_net():
    net = gluon.nn.Sequential()
    net.add(nn.Dense(10, activation='relu'),
            nn.Dense(1))
    net.initialize(init.Xavier())
    return net

# Least mean squares loss
loss = gluon.loss.L2Loss()
```

Now we are ready to train.

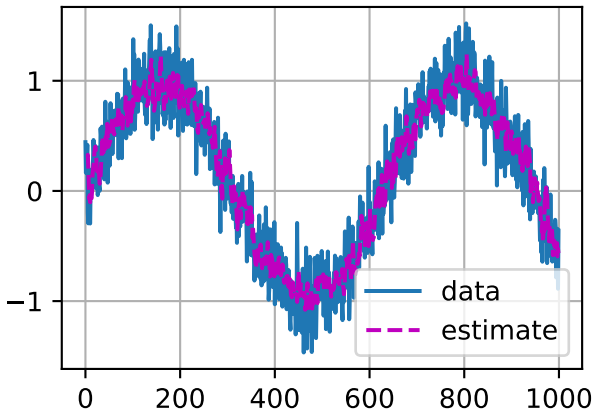
```
def train_net(net, train_iter, loss, epochs, lr):
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    for epoch in range(1, epochs + 1):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        print('epoch %d, loss: %f' % (
            epoch, d2l.evaluate_loss(net, train_iter, loss)))

net = get_net()
train_net(net, train_iter, loss, 10, 0.01)
```

```
epoch 1, loss: 0.037273
epoch 2, loss: 0.030691
epoch 3, loss: 0.030349
epoch 4, loss: 0.028200
epoch 5, loss: 0.028039
epoch 6, loss: 0.029394
epoch 7, loss: 0.028047
epoch 8, loss: 0.028731
epoch 9, loss: 0.028676
epoch 10, loss: 0.027363
```

The both training and test loss are small and we would expect our model to work well. Let us see what this means in practice. The first thing to check is how well the model is able to predict what happens in the next timestep.

```
estimates = net(features)
d2l.plot([time, time[tau:]], [x, estimates],
         legend=['data', 'estimate'])
```



10.1.3 Predictions

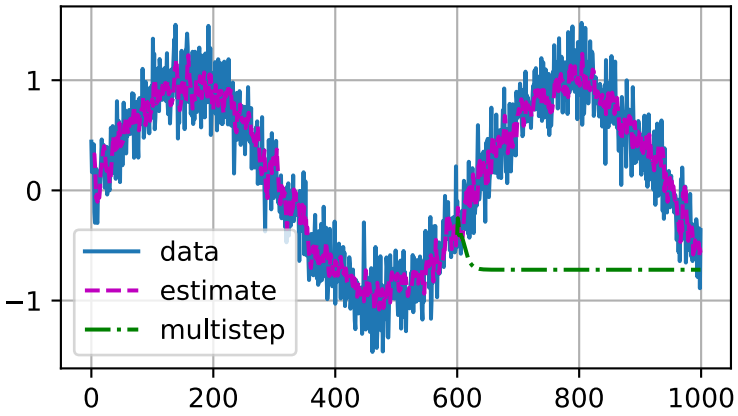
This looks nice, just as we expected it. Even beyond 600 observations the estimates still look rather trustworthy. There is just one little problem to this - if we observe data only until time step 600, we cannot hope to receive the ground truth

for all future predictions. Instead, we need to work our way forward one step at a time:

$$\begin{aligned} x_{601} &= f(x_{600}, \ldots, x_{597}) \\ x_{602} &= f(x_{601}, \ldots, x_{598}) \\ x_{603} &= f(x_{602}, \ldots, x_{599}) \end{aligned} \tag{10.1.6}$$

In other words, very quickly will we have to use our own predictions to make future predictions. Let us see how well this goes.

```
predictions = np.zeros(T)
predictions[:n_train] = x[:n_train]
for i in range(n_train, T):
    predictions[i] = net(
        predictions[(i-tau):i].reshape(1,-1)).reshape(1)
d2l.plot([time, time[tau:], time[n_train:]],
         [x, estimates, predictions[n_train:]],
         legend=['data', 'estimate', 'multistep'], figsize=(4.5, 2.5))
```



As the above example shows, this is a spectacular failure. The estimates decay to a constant pretty quickly after a few prediction steps. Why did the algorithm work so poorly? This is ultimately due to the fact that errors build up. Let us say that after step 1 we have some error $\epsilon_1 = \bar{\epsilon}$. Now the *input* for step 2 is perturbed by ϵ_1, hence we suffer some error in the order of $\epsilon_2 = \bar{\epsilon} + L\epsilon_1$, and so on. The error can diverge rather rapidly from the true observations. This is a common phenomenon - for instance weather forecasts for the next 24 hours tend to be pretty accurate but beyond that their accuracy declines rapidly. We will discuss methods for improving this throughout this chapter and beyond.

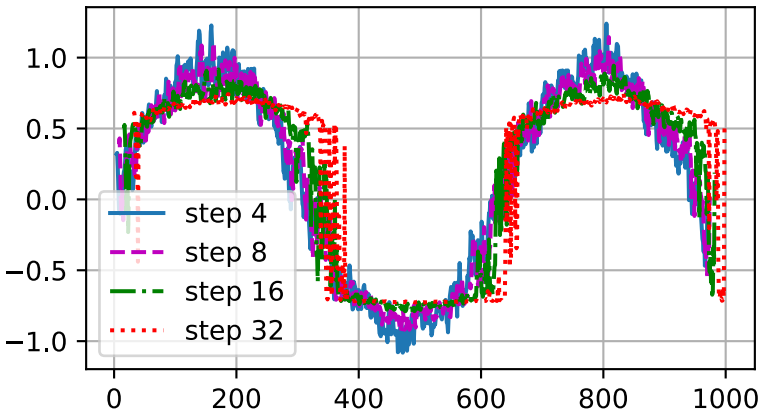
Let us verify this observation by computing the k-step predictions on the entire sequence.

```
k = 33  # Look up to k - tau steps ahead

features = np.zeros((k, T-k))
for i in range(tau):  # Copy the first tau features from x
    features[i] = x[i:T-k+i]

for i in range(tau, k):  # Predict the (i-tau)-th step
    features[i] = net(features[(i-tau):i].T).T

steps = (4, 8, 16, 32)
d2l.plot([time[i:T-k+i] for i in steps], [features[i] for i in steps],
         legend=['step %d'%i for i in steps], figsize=(4.5, 2.5))
```



This clearly illustrates how the quality of the estimates changes as we try to predict further into the future. While the 8-step predictions are still pretty good, anything beyond that is pretty useless.

10.1.4 Summary

- Sequence models require specialized statistical tools for estimation. Two popular choices are autoregressive models and latent-variable autoregressive models.
- As we predict further in time, the errors accumulate and the quality of the estimates degrades, often dramatically.
- There is quite a difference in difficulty between filling in the blanks in a sequence (smoothing) and forecasting. Consequently, if you have a time series, always respect the temporal order of the data when training, i.e., never train on future data.
- For causal models (e.g., time going forward), estimating the forward direction is typically a lot easier than the reverse direction, i.e., we can get by with simpler networks.

10.1.5 Exercises

1. Improve the above model.
    - Incorporate more than the past 4 observations? How many do you really need?
    - How many would you need if there were no noise? Hint - you can write sin and cos as a differential equation.
    - Can you incorporate older features while keeping the total number of features constant? Does this improve accuracy? Why?
    - Change the architecture and see what happens.
2. An investor wants to find a good security to buy. She looks at past returns to decide which one is likely to do well. What could possibly go wrong with this strategy?
3. Does causality also apply to text? To which extent?
4. Give an example for when a latent variable autoregressive model might be needed to capture the dynamic of the data.

10.1.6 Scan the QR Code to Discuss[130]

10.2 Text Preprocessing

Text is an important example of sequence data. An article can be simply viewed as a sequence of words, or a sequence of characters. Given text data is a major data format besides images we are using in this book, this section will dedicate to explain the common preprocessing steps for text data. Such preprocessing often consists of four steps:

1. Loads texts as strings into memory.
2. Splits strings into tokens, a token could be a word or a character.
3. Builds a vocabulary for these tokens to map them into numerical indices.
4. Maps all tokens in the data into indices to facilitate to feed into models.

10.2.1 Data Loading

To get started we load text from H. G. Wells' Time Machine[131]. This is a fairly small corpus of just over 30,000 words but for the purpose of what we want to illustrate this is just fine. More realistic document collections contain many billions of words. The following function read the dataset into a list of sentences, each sentence is a string. Here we ignore punctuation and capitalization.

```
import collections
import re

# Saved in the d2l package for later use
def read_time_machine():
    """Load the time machine book into a list of sentences."""
    with open('../data/timemachine.txt', 'r') as f:
        lines = f.readlines()
    return [re.sub('[^A-Za-z]+', ' ', line.strip().lower())
            for line in lines]

lines = read_time_machine()
'# sentences %d' % len(lines)
```

```
'# sentences 3221'
```

10.2.2 Tokenization

For each sentence, we split it into a list of tokens. A token is a data point the model will train and predict. The following function supports split a sentence into words or characters, and return a list of split sentences.

[130] https://discuss.mxnet.io/t/2860
[131] http://www.gutenberg.org/ebooks/35

```
# Saved in the d2l package for later use
def tokenize(lines, token='word'):
    """Split sentences into word or char tokens"""
    if token == 'word':
        return [line.split(' ') for line in lines]
    elif token == 'char':
        return [list(line) for line in lines]
    else:
        print('ERROR: unkown token type '+token)

tokens = tokenize(lines)
tokens[0:2]
```

```
[['the', 'time', 'machine', 'by', 'h', 'g', 'wells', ''], ['']]
```

10.2.3 Vocabulary

The string type of the token is inconvienet to be used by models, which take numerical inputs. Now let us build a dictionary, often called *vocabulary* as well, to map string tokens into numerical indices starting from 0. To do so, we first count the unique tokens in all documents, called *corpus*, and then assign a numerical index to each unique token according to its frequency. Rarely appeared tokens are often removed to reduce the complexity. A token does not exist in corpus or has been removed is mapped into a special unknown ("<unk>") token. We optionally add another three special tokens: "<pad>" a token for padding, "<bos>" to present the beginning for a sentence, and "<eos>" for the ending of a sentence.

```
# Saved in the d2l package for later use
class Vocab(object):
    def __init__(self, tokens, min_freq=0, use_special_tokens=False):
        # Sort according to frequencies
        counter = count_corpus(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[0])
        self.token_freqs.sort(key=lambda x: x[1], reverse=True)
        if use_special_tokens:
            # padding, begin of sentence, end of sentence, unknown
            self.pad, self.bos, self.eos, self.unk = (0, 1, 2, 3)
            uniq_tokens = ['<pad>', '<bos>', '<eos>', '<unk>']
        else:
            self.unk, uniq_tokens = 0, ['<unk>']
        uniq_tokens +=  [token for token, freq in self.token_freqs
                         if freq >= min_freq and token not in uniq_tokens]
        self.idx_to_token, self.token_to_idx = [], dict()
        for token in uniq_tokens:
            self.idx_to_token.append(token)
            self.token_to_idx[token] = len(self.idx_to_token) - 1

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
```

(continues on next page)

(continued from previous page)

```
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]

# Saved in the d2l package for later use
def count_corpus(sentences):
    # Flatten a list of token lists into a list of tokens
    tokens = [tk for line in sentences for tk in line]
    return collections.Counter(tokens)
```

We construct a vocabulary with the time machine dataset as the corpus, and then print the map between a few tokens to indices.

```
vocab = Vocab(tokens)
print(list(vocab.token_to_idx.items())[0:10])
```

```
[('<unk>', 0), ('the', 1), ('', 2), ('i', 3), ('and', 4), ('of', 5), ('a', 6), ('to',␣
↪7), ('was', 8), ('in', 9)]
```

After that, we can convert each sentence into a list of numerical indices. To illustrate things we print two sentences with their corresponding indices.

```
for i in range(8, 10):
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])
```

```
words: ['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'convenient', 'to
↪', 'speak', 'of', 'him', '']
indices: [1, 20, 72, 17, 38, 12, 120, 43, 706, 7, 660, 5, 112, 2]
words: ['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', 'grey',
↪'eyes', 'shone', 'and']
indices: [8, 1654, 6, 3864, 634, 7, 131, 26, 344, 127, 484, 4]
```

10.2.4 Put All Things Together

We packaged the above code in the `load_corpus_time_machine` function, which returns `corpus`, a list of token indices, and `vocab`, the vocabulary. The modification we did here is that `corpus` is a single list, not a list of token lists, since we do not the sequence information in the following models. Besides, we use character tokens to simplify the training in later sections.

```
# Saved in the d2l package for later use
def load_corpus_time_machine(max_tokens=-1):
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)
    corpus = [vocab[tk] for line in tokens for tk in line]
    if max_tokens > 0: corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

```
(171489, 28)
```

10.2.5 Summary

- Documents are preprocessed by tokenizing the words or characters and mapping them into indices.

10.2.6 Exercises

- Tokenization is a key preprocessing step. It varies for different languages. Try to find another 3 commonly used methods to tokenize sentences.

10.2.7 Scan the QR Code to Discuss[132]

10.3 Language Models and Data Sets

In Section 10.2, we see how to map text data into tokens, and these tokens can be viewed as a time series of discrete observations. Assuming the tokens in a text of length T are in turn $x_1, x_2, \ldots, x_T$, then, in the discrete time series, $x_t (1 \leq t \leq T)$ can be considered as the output or label of time step t. Given such a sequence, the goal of a language model is to estimate the probability

$$p(x_1, x_2, \ldots, x_T). \tag{10.3.1}$$

Language models are incredibly useful. For instance, an ideal language model would be able to generate natural text just on its own, simply by drawing one word at a time $w_t \sim p(w_t \mid w_{t-1}, \ldots, w_1)$. Quite unlike the monkey using a typewriter, all text emerging from such a model would pass as natural language, e.g., English text. Furthermore, it would be sufficient for generating a meaningful dialog, simply by conditioning the text on previous dialog fragments. Clearly we are still very far from designing such a system, since it would need to understand the text rather than just generate grammatically sensible content.

Nonetheless language models are of great service even in their limited form. For instance, the phrases *'to recognize speech'* and *'to wreck a nice beach'* sound very similar. This can cause ambiguity in speech recognition, ambiguity that is easily resolved through a language model which rejects the second translation as outlandish. Likewise, in a document summarization algorithm it is worth while knowing that *'dog bites man'* is much more frequent than *'man bites dog'*, or that *'let us eat grandma'* is a rather disturbing statement, whereas *'let us eat, grandma'* is much more benign.

[132] https://discuss.mxnet.io/t/2363

10.3.1 Estimating a language model

The obvious question is how we should model a document, or even a sequence of words. Recall the analysis we applied to sequence models in the previous section, we can start by applying basic probability rules:

$$p(w_1, w_2, \ldots, w_T) = p(w_1) \prod_{t=2}^{T} p(w_t \mid w_1, \ldots, w_{t-1}). \tag{10.3.2}$$

For example, the probability of a text sequence containing four tokens consisting of words and punctuation would be given as:

$$p(\text{Statistics}, \text{is}, \text{fun}, .) = p(\text{Statistics})p(\text{is} \mid \text{Statistics})p(\text{fun} \mid \text{Statistics}, \text{is})p(. \mid \text{Statistics}, \text{is}, \text{fun}). \tag{10.3.3}$$

In order to compute the language model, we need to calculate the probability of words and the conditional probability of a word given the previous few words, i.e., the language model parameters. Here, we assume that the training dataset is a large text corpus, such as all Wikipedia entries, Project Gutenberg[133], or all text posted online on the web. The probability of words can be calculated from the relative word frequency of a given word in the training dataset.

For example, $p(\text{Statistics})$ can be calculated by the probability of any sentence starting with the word 'statistics'. A slightly less accurate approach would be to count all occurrences of the word 'statistics' and divide it by the total number of words in the corpus. This works fairly well, particularly for frequent words. Moving on, we could attempt to estimate

$$\hat{p}(\text{is} \mid \text{Statistics}) = \frac{n(\text{Statistics is})}{n(\text{Statistics})}. \tag{10.3.4}$$

Here $n(w)$ and $n(w, w')$ are the number of occurrences of singletons and pairs of words respectively. Unfortunately, estimating the probability of a word pair is somewhat more difficult, since the occurrences of *'Statistics is'* are a lot less frequent. In particular, for some unusual word combinations it may be tricky to find enough occurrences to get accurate estimates. Things take a turn for the worse for 3-word combinations and beyond. There will be many plausible 3-word combinations that we likely will not see in our dataset. Unless we provide some solution to give such word combinations nonzero weight, we will not be able to use these as a language model. If the dataset is small or if the words are very rare, we might not find even a single one of them.

A common strategy is to perform some form of Laplace smoothing. We already encountered this in our discussion of naive bayes in Section 18.9 where the solution was to add a small constant to all counts. This helps with singletons, e.g., via

$$\begin{aligned}
\hat{p}(w) &= \frac{n(w) + \epsilon_1/m}{n + \epsilon_1} \\
\hat{p}(w' \mid w) &= \frac{n(w, w') + \epsilon_2 \hat{p}(w')}{n(w) + \epsilon_2} \\
\hat{p}(w'' \mid w', w) &= \frac{n(w, w', w'') + \epsilon_3 \hat{p}(w', w'')}{n(w, w') + \epsilon_3}
\end{aligned} \tag{10.3.5}$$

Here the coefficients $\epsilon_i > 0$ determine how much we use the estimate for a shorter sequence as a fill-in for longer ones. Moreover, m is the total number of words we encounter. The above is a rather primitive variant of what is Kneser-Ney smoothing and Bayesian Nonparametrics can accomplish. See e.g., (Wood et al., 2011) for more details of how to accomplish this. Unfortunately, models like this get unwieldy rather quickly for the following reasons. First, we need to store all counts. Secondly, this entirely ignores the meaning of the words. For instance, *'cat'* and *'feline'* should occur in related contexts. It is quite difficult to adjust such models to additional context, whereas, deep learning based language models are well suited to take this into account. Lastly, long word sequences are almost certain to be novel, hence a model that simply counts the frequency of previously seen word sequences is bound to perform poorly there.

[133] https://en.wikipedia.org/wiki/Project_Gutenberg

10.3.2 Markov Models and n-grams

Before we discuss solutions involving deep learning, we need some more terminology and concepts. Recall our discussion of Markov Models in the previous section, let us apply this to language modeling. A distribution over sequences satisfies the Markov property of first order if $p(w_{t+1} \mid w_t, \ldots, w_1) = p(w_{t+1} \mid w_t)$. Higher orders correspond to longer dependencies. This leads to a number of approximations that we could apply to model a sequence:

$$\begin{aligned} p(w_1, w_2, w_3, w_4) &= p(w_1)p(w_2)p(w_3)p(w_4) \\ p(w_1, w_2, w_3, w_4) &= p(w_1)p(w_2 \mid w_1)p(w_3 \mid w_2)p(w_4 \mid w_3) \\ p(w_1, w_2, w_3, w_4) &= p(w_1)p(w_2 \mid w_1)p(w_3 \mid w_1, w_2)p(w_4 \mid w_2, w_3) \end{aligned} \tag{10.3.6}$$

The probability formula that involves one, two, and three variables are typically referred to as unigram, bigram, and trigram models respectively. In the following, we will learn how to design better models.

10.3.3 Natural Language Statistics

Let us see how this works on real data. We construct a vocabulary based on the time machine data similar to Section 10.2 and print the top-10 most frequent words.

```
import d2l
from mxnet import np, npx
import random
npx.set_np()

tokens = d2l.tokenize(d2l.read_time_machine())
vocab = d2l.Vocab(tokens)
print(vocab.token_freqs[:10])
```

```
[('the', 2261), ('', 1282), ('i', 1267), ('and', 1245), ('of', 1155), ('a', 816), ('to
↪', 695), ('was', 552), ('in', 541), ('that', 443)]
```

As we can see, the most popular words are actually quite boring to look at. They are often referred to as stop words[134] and thus filtered out. That said, they still carry meaning and we will use them nonetheless. However, one quite clear thing is that the word frequency decays rather rapidly. The 10th most frequent word is less than $1/5$ as common as the most popular one. To get a better idea we plot the graph of the word frequencies.

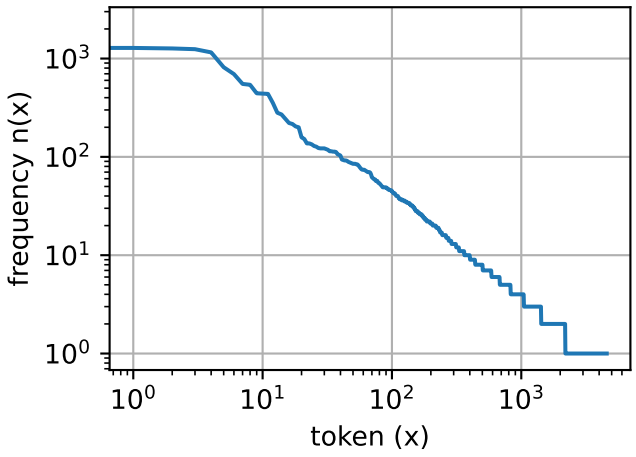
```
freqs = [freq for token, freq in vocab.token_freqs]
d2l.plot(freqs, xlabel='token (x)', ylabel='frequency n(x)',
         xscale='log', yscale='log')
```

We are on to something quite fundamental here - the word frequencies decay rapidly in a well defined way. After dealing with the first four words as exceptions ('the', 'i', 'and', 'of'), all remaining words follow a straight line on a log-log plot. This means that words satisfy Zipf's law[135] which states that the item frequency is given by

$$n(x) \propto (x + c)^{-\alpha} \text{ and hence } \log n(x) = -\alpha \log(x + c) + \text{const.} \tag{10.3.7}$$

This should already give us pause if we want to model words by count statistics and smoothing. After all, we will significantly overestimate the frequency of the tail, aka the infrequent words. But what about the other word combinations (such as bigrams, trigrams, and beyond)? Let us see whether the bigram frequencies behave in the same manner as the unigram frequencies.

[134] https://en.wikipedia.org/wiki/Stop_words
[135] https://en.wikipedia.org/wiki/Zipf%27s_law




```
bigram_tokens = [[pair for pair in zip(line[:-1], line[1:])] for line in tokens]
bigram_vocab = d2l.Vocab(bigram_tokens)
print(bigram_vocab.token_freqs[:10])
```

```
[(('of', 'the'), 297), (('in', 'the'), 161), (('i', 'had'), 126), (('and', 'the'),␣
↪104), (('i', 'was'), 104), (('the', 'time'), 97), (('it', 'was'), 94), (('to', 'the
↪'), 81), (('as', 'i'), 75), (('of', 'a'), 69)]
```

Two things are notable. Out of the 10 most frequent word pairs, 9 are composed of stop words and only one is relevant to the actual book - 'the time'. Furthermore, let us see whether the trigram frequencies behave in the same manner.

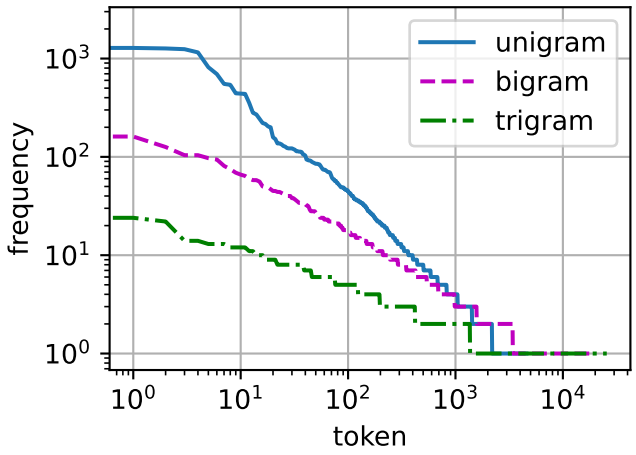
```
trigram_tokens = [[triple for triple in zip(line[:-2], line[1:-1], line[2:])]
                  for line in tokens]
trigram_vocab = d2l.Vocab(trigram_tokens)
print(trigram_vocab.token_freqs[:10])
```

```
[(('the', 'time', 'traveller'), 53), (('the', 'time', 'machine'), 24), (('the',
↪'medical', 'man'), 22), (('it', 'seemed', 'to'), 14), (('it', 'was', 'a'), 14), (('i
↪', 'began', 'to'), 13), (('i', 'did', 'not'), 13), (('i', 'saw', 'the'), 13), ((
↪'here', 'and', 'there'), 12), (('i', 'could', 'see'), 12)]
```

Last, let us visualize the token frequencies among these three gram models: unigram, bigram, and trigram.

```
bigram_freqs = [freq for token, freq in bigram_vocab.token_freqs]
trigram_freqs = [freq for token, freq in trigram_vocab.token_freqs]
d2l.plot([freqs, bigram_freqs, trigram_freqs], xlabel='token',
         ylabel='frequency', xscale='log', yscale='log',
         legend=['unigram', 'bigram', 'trigram'])
```

The graph is quite exciting for a number of reasons. Firstly, beyond unigram words, also sequences of words appear to be following Zipf's law, albeit with a lower exponent, depending on the sequence length. Secondly, the number of distinct n-grams is not that large. This gives us hope that there is quite a lot of structure in language. Thirdly, many n-grams occur very rarely, which makes Laplace smoothing rather unsuitable for language modeling. Instead, we will use deep learning based models.



10.3.4 Training Data Preparation

Before introducing the model, let us assume we will use a neural network to train a language model. Now the question is how to read minibatches of examples and labels at random. Since sequence data is by its very nature sequential, we need to address the issue of processing it. We did so in a rather ad-hoc manner when we introduced in Section 10.1. Let us formalize this a bit.

In Fig. 10.3.1, we visualized several possible ways to obtain 5-grams in a sentence, here a token is a character. Note that we have quite some freedom since we could pick an arbitrary offset.

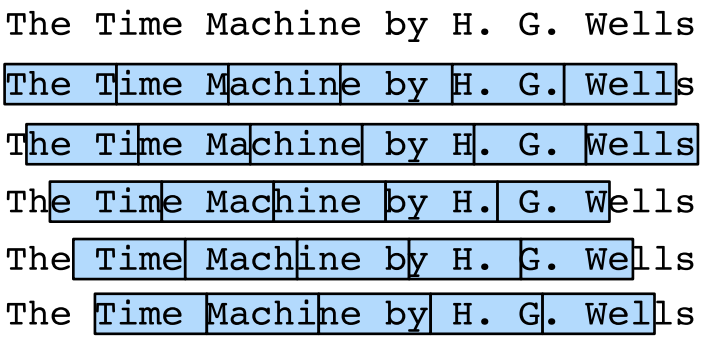


Fig. 10.3.1: Different offsets lead to different subsequences when splitting up text.

In fact, any one of these offsets is fine. Hence, which one should we pick? In fact, all of them are equally good. But if we pick all offsets we end up with rather redundant data due to overlap, particularly if the sequences are long. Picking just a random set of initial positions is no good either since it does not guarantee uniform coverage of the array. For instance, if we pick n elements at random out of a set of n with random replacement, the probability for a particular element not being picked is $(1 - 1/n)^n \to e^{-1}$. This means that we cannot expect uniform coverage this way. Even randomly permuting a set of all offsets does not offer good guarantees. Instead we can use a simple trick to get both *coverage* and *randomness*: use a random offset, after which one uses the terms sequentially. We describe how to accomplish this for both random sampling and sequential partitioning strategies below.

Random Sampling

The following code randomly generates a minibatch from the data each time. Here, the batch size `batch_size` indicates to the number of examples in each minibatch and `num_steps` is the length of the sequence (or time steps if we have a

time series) included in each example. In random sampling, each example is a sequence arbitrarily captured on the original sequence. The positions of two adjacent random minibatches on the original sequence are not necessarily adjacent. The target is to predict the next character based on what we have seen so far, hence the labels are the original sequence, shifted by one character.

```
# Saved in the d2l package for later use
def seq_data_iter_random(corpus, batch_size, num_steps):
    # Offset the iterator over the data for uniform starts
    corpus = corpus[random.randint(0, num_steps):]
    # Subtract 1 extra since we need to account for label
    num_examples = ((len(corpus) - 1) // num_steps)
    example_indices = list(range(0, num_examples * num_steps, num_steps))
    random.shuffle(example_indices)
    # This returns a sequence of the length num_steps starting from pos
    data = lambda pos: corpus[pos: pos + num_steps]
    # Discard half empty batches
    num_batches = num_examples // batch_size
    for i in range(0, batch_size * num_batches, batch_size):
        # Batch_size indicates the random examples read each time
        batch_indices = example_indices[i:(i+batch_size)]
        X = [data(j) for j in batch_indices]
        Y = [data(j + 1) for j in batch_indices]
        yield np.array(X), np.array(Y)
```

Let us generate an artificial sequence from 0 to 30. We assume that the batch size and numbers of time steps are 2 and 5 respectively. This means that depending on the offset we can generate between 4 and 5 (x, y) pairs. With a minibatch size of 2, we only get 2 minibatches.

```
my_seq = list(range(30))
for X, Y in seq_data_iter_random(my_seq, batch_size=2, num_steps=6):
    print('X: ', X, '\nY:', Y)
```

```
X:  [[ 0.  1.  2.  3.  4.  5.]
 [ 6.  7.  8.  9. 10. 11.]]
Y: [[ 1.  2.  3.  4.  5.  6.]
 [ 7.  8.  9. 10. 11. 12.]]
X:  [[18. 19. 20. 21. 22. 23.]
 [12. 13. 14. 15. 16. 17.]]
Y: [[19. 20. 21. 22. 23. 24.]
 [13. 14. 15. 16. 17. 18.]]
```

Sequential partitioning

In addition to random sampling of the original sequence, we can also make the positions of two adjacent random minibatches adjacent in the original sequence.

```
# Saved in the d2l package for later use
def seq_data_iter_consecutive(corpus, batch_size, num_steps):
    # Offset for the iterator over the data for uniform starts
    offset = random.randint(0, num_steps)
    # Slice out data - ignore num_steps and just wrap around
    num_indices = ((len(corpus) - offset - 1) // batch_size) * batch_size
    Xs = np.array(corpus[offset:offset+num_indices])
    Ys = np.array(corpus[offset+1:offset+1+num_indices])
```

(continues on next page)

(continued from previous page)

```
    Xs, Ys = Xs.reshape(batch_size, -1), Ys.reshape(batch_size, -1)
    num_batches = Xs.shape[1] // num_steps
    for i in range(0, num_batches * num_steps, num_steps):
        X = Xs[:,i:(i+num_steps)]
        Y = Ys[:,i:(i+num_steps)]
        yield X, Y
```

Using the same settings, print input `X` and label `Y` for each minibatch of examples read by random sampling. The positions of two adjacent minibatches on the original sequence are adjacent.

```
for X, Y in seq_data_iter_consecutive(my_seq, batch_size=2, num_steps=6):
    print('X: ', X, '\nY:', Y)
```

```
X:  [[ 2.  3.  4.  5.  6.  7.]
 [15. 16. 17. 18. 19. 20.]]
Y: [[ 3.  4.  5.  6.  7.  8.]
 [16. 17. 18. 19. 20. 21.]]
X:  [[ 8.  9. 10. 11. 12. 13.]
 [21. 22. 23. 24. 25. 26.]]
Y: [[ 9. 10. 11. 12. 13. 14.]
 [22. 23. 24. 25. 26. 27.]]
```

Now we wrap the above two sampling functions to a class so that we can use it as a Gluon data iterator later.

```
# Saved in the d2l package for later use
class SeqDataLoader(object):
    """A iterator to load sequence data"""
    def __init__(self, batch_size, num_steps, use_random_iter, max_tokens):
        if use_random_iter:
            data_iter_fn = d2l.seq_data_iter_random
        else:
            data_iter_fn = d2l.seq_data_iter_consecutive
        self.corpus, self.vocab = d2l.load_corpus_time_machine(max_tokens)
        self.get_iter = lambda: data_iter_fn(self.corpus, batch_size, num_steps)

    def __iter__(self):
        return self.get_iter()
```

Lastly, we define a function `load_data_time_machine` that returns both the data iterator and the vocabulary, so we can use it similarly as other functions with `load_data` prefix.

```
# Saved in the d2l package for later use
def load_data_time_machine(batch_size, num_steps, use_random_iter=False,
                           max_tokens=10000):
    data_iter = SeqDataLoader(
        batch_size, num_steps, use_random_iter, max_tokens)
    return data_iter, data_iter.vocab
```

10.3.5 Summary

- Language models are an important technology for natural language processing.
- n-grams models make it convenient to deal with long sequences by truncating the dependence.

- Long sequences suffer from the problem that they occur very rarely or never.
- Zipf's law governs the word distribution for not only unigrams but also the other n-grams.
- There is a lot of structure but not enough frequency to deal with infrequent word combinations efficiently via Laplace smoothing.
- The main choices for sequence partitioning are picking between consecutive and random sequences.
- Given the overall document length, it is usually acceptable to be slightly wasteful with the documents and discard half-empty minibatches.

10.3.6 Exercises

1. Suppose there are 100,000 words in the training dataset. How many word frequencies and multi-word adjacent frequencies does a four-gram need to store?
2. Review the smoothed probability estimates. Why are they not accurate? Hint - we are dealing with a contiguous sequence rather than singletons.
3. How would you model a dialogue?
4. Estimate the exponent of Zipf's law for unigrams, bigrams and trigrams.
5. What other minibatch data sampling methods can you think of?
6. Why is it a good idea to have a random offset?
    - Does it really lead to a perfect uniform distribution over the sequences on the document?
    - What would you have to do to make things even more uniform?
7. If we want a sequence example to be a complete sentence, what kinds of problems does this introduce in minibatch sampling? Why would we want to do this anyway?

10.3.7 Scan the QR Code to Discuss[136]

10.4 Recurrent Neural Networks

In Section 10.3 we introduced n-gram models, where the conditional probability of word x_t at position t only depends on the $n-1$ previous words. If we want to check the possible effect of words earlier than $t-(n-1)$ on x_t, we need to increase n. However, the number of model parameters would also increase exponentially with it, as we need to store $|V|^n$ numbers for a vocabulary V. Hence, rather than modeling $p(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$ it is preferable to use a *latent variable model* in which we have

$$p(x_t \mid x_{t-1}, \ldots, x_1) \approx p(x_t \mid x_{t-1}, h_t). \tag{10.4.1}$$

[136] https://discuss.mxnet.io/t/2361

Here h_t is a *latent variable* that stores the sequence information. A latent variable is also called as *hidden variable*, *hidden state* or *hidden state variable*. The hidden state at time t could be computed based on both input x_t and hidden state h_{t-1}, that is

$$h_t = f(x_t, h_{t-1}). \tag{10.4.2}$$

For a sufficiently powerful function f, the latent variable model is not an approximation. After all, h_t could simply store all the data it observed so far. We discussed this in Section 10.1. But it could potentially makes both computation and storage expensive.

Note that we also use h to denote by the number of hidden units of a hidden layer. Hidden layers and hidden states refer to two very different concepts. Hidden layers are, as explained, layers that are hidden from view on the path from input to output. Hidden states are technically speaking *inputs* to whatever we do at a given step. Instead, they can only be computed by looking at data at previous iterations. In this sense they have much in common with latent variable models in statistics, such as clustering or topic models where the clusters affect the output but cannot be directly observed.

Recurrent neural networks are neural networks with hidden states. Before introducing this model, let us first revisit the multi-layer perceptron introduced in Section 6.1.

10.4.1 Recurrent Networks Without Hidden States

Let us take a look at a multilayer perceptron with a single hidden layer. Given a minibatch of the instances $\mathbf{X} \in \mathbb{R}^{n \times d}$ with sample size n and d inputs. Let the hidden layer's activation function be ϕ. Hence, the hidden layer's output $\mathbf{H} \in \mathbb{R}^{n \times h}$ is calculated as

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h). \tag{10.4.3}$$

Here, we have the weight parameter $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, bias parameter $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$, and the number of hidden units h, for the hidden layer.

The hidden variable $\mathbf{H}$ is used as the input of the output layer. The output layer is given by

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q. \tag{10.4.4}$$

Here, $\mathbf{O} \in \mathbb{R}^{n \times q}$ is the output variable, $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ is the weight parameter, and $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ is the bias parameter of the output layer. If it is a classification problem, we can use softmax($\mathbf{O}$) to compute the probability distribution of the output category.

This is entirely analogous to the regression problem we solved previously in Section 10.1, hence we omit details. Suffice it to say that we can pick (x_t, x_{t-1}) pairs at random and estimate the parameters $\mathbf{W}$ and $\mathbf{b}$ of our network via autograd and stochastic gradient descent.

10.4.2 Recurrent Networks with Hidden States

Matters are entirely different when we have hidden states. Let us look at the structure in some more detail. Remember that we often call iteration t as time t in an optimization algorithm, time in a recurrent neural network refers to steps within an iteration. Assume that we have $\mathbf{X}_t \in \mathbb{R}^{n \times d}$, $t = 1, \ldots, T$, in an iteration. And $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ is the hidden variable of time step t from the sequence. Unlike the multilayer perceptron, here we save the hidden variable $\mathbf{H}_{t-1}$ from the previous time step and introduce a new weight parameter $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$, to describe how to use the hidden variable of the previous time step in the current time step. Specifically, the calculation of the hidden variable of the current time step is determined by the input of the current time step together with the hidden variable of the previous time step:

$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h). \tag{10.4.5}$$

Compared with (10.4.3), we added one more $\mathbf{H}_{t-1}\mathbf{W}_{hh}$ here. From the relationship between hidden variables $\mathbf{H}_t$ and $\mathbf{H}_{t-1}$ of adjacent time steps, we know that those variables captured and retained the sequence's historical information up

to the current time step, just like the state or memory of the neural network's current time step. Therefore, such a hidden variable is called a *hidden state*. Since the hidden state uses the same definition of the previous time step in the current time step, the computation of the equation above is recurrent, hence the name recurrent neural network (RNN).

There are many different RNN construction methods. RNNs with a hidden state defined by the equation above are very common. For time step t, the output of the output layer is similar to the computation in the multilayer perceptron:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q \tag{10.4.6}$$

RNN parameters include the weight $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ of the hidden layer with the bias $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$, and the weight $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ of the output layer with the bias $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$. It is worth mentioning that RNNs always use these model parameters, even for different time steps. Therefore, the number of RNN model parameters does not grow as the number of time steps increases.

Fig. 10.4.1 shows the computational logic of an RNN at three adjacent time steps. In time step t, the computation of the hidden state can be treated as an entry of a fully connected layer with the activation function ϕ after concatenating the input $\mathbf{X}_t$ with the hidden state $\mathbf{H}_{t-1}$ of the previous time step. The output of the fully connected layer is the hidden state of the current time step $\mathbf{H}_t$. Its model parameter is the concatenation of $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$, with a bias of $\mathbf{b}_h$. The hidden state of the current time step t, $\mathbf{H}_t$, will participate in computing the hidden state $\mathbf{H}_{t+1}$ of the next time step $t+1$. What is more, $\mathbf{H}_t$ will become the input for $\mathbf{O}_t$, the fully connected output layer of the current time step.

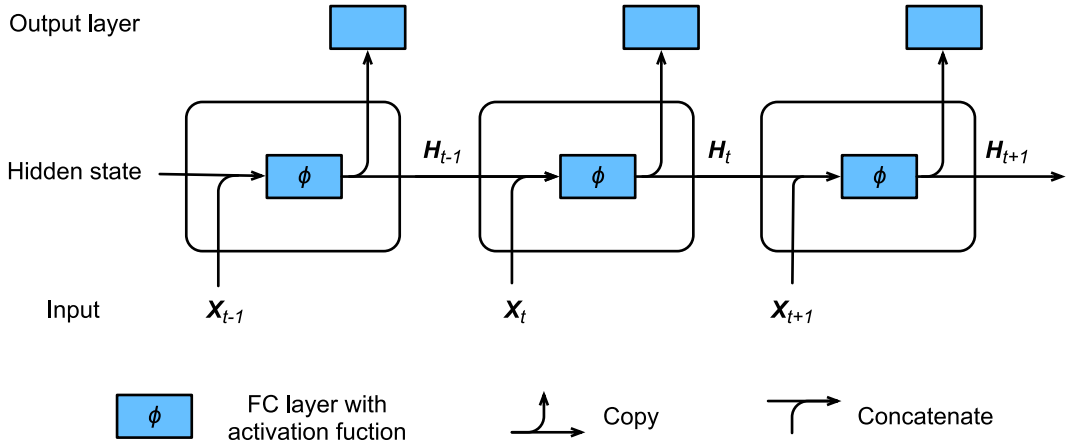


Fig. 10.4.1: An RNN with a hidden state.

10.4.3 Steps in a Language Model

Now we illustrate how RNNs can be used to build a language model. For simplicity of illustration we use words rather than characters as the inputs, since the former are easier to comprehend. Let the number of minibatch size be 1, and the sequence of the text be the beginning of our dataset, i.e., "The Time Machine by H. G. Wells". The figure below illustrates how to estimate the next word based on the present and previous words. During the training process, we run a softmax operation on the output from the output layer for each time step, and then use the cross-entropy loss function to compute the error between the result and the label. Due to the recurrent computation of the hidden state in the hidden layer, the output of time step 3, $\mathbf{O}_3$, is determined by the text sequence "the", "time", and "machine" respectively. Since the next word of the sequence in the training data is "by", the loss of time step 3 will depend on the probability distribution of the next word generated based on the feature sequence "the", "time", "machine" and the label "by" of this time step.

In practice, each word is presented by a d dimensional vector, and we use a batch size $n > 1$. Therefore, the input $\mathbf{X}_t$ at time step t will be a $n \times d$ matrix, which is identical to what we discussed before.

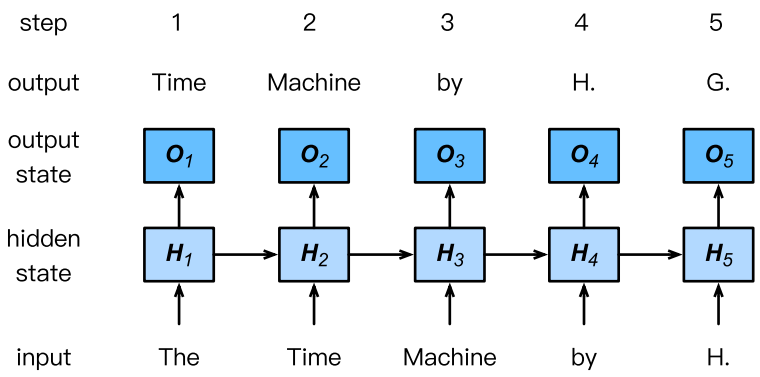


Fig. 10.4.2: Word-level RNN language model. The input and label sequences are `The Time Machine by H.` and `Time Machine by H. G.` respectively.

10.4.4 Perplexity

Last, let us discuss about how to measure the sequence model quality. One way is to check how surprising the text is. A good language model is able to predict with high accuracy tokens that what we will see next. Consider the following continuations of the phrase `"It is raining"`, as proposed by different language models:

1. `It is raining outside`
2. `It is raining banana tree`
3. `It is raining piouw;kcj pwepoiut`

In terms of quality, example 1 is clearly the best. The words are sensible and logically coherent. While it might not quite accurately reflect which word follows semantically (`in San Francisco` and `in winter` would have been perfectly reasonable extensions), the model is able to capture the kind of meaningful words. Example 2 is considerably worse by producing a nonsensical and borderline dysgrammatical extension. Nonetheless, at least the model has learned how to spell words and some degree of correlation between words. Lastly, example 3 indicates a poorly trained model that does not fit data properly.

We might measure the quality of the model by computing $p(w)$, i.e., the likelihood of the sequence. Unfortunately this is a number that is hard to understand and difficult to compare. After all, shorter sequences are much more likely to occur than the longer ones, hence evaluating the model on Tolstoy's magnum opus 'War and Peace'[137] will inevitably produce a much smaller likelihood than, say, on Saint-Exupery's novella 'The Little Prince'[138]. What is missing is the equivalent of an average.

Information theory comes handy here and we will introduce more in Section 18.11. If we want to compress text, we can ask about estimating the next symbol given the current set of symbols. A lower bound on the number of bits is given by $-\log_2 p(x_t \mid x_{t-1}, \ldots, x_1)$. A good language model should allow us to predict the next word quite accurately. Thus, it should allow us to spend very few bits on compressing the sequence. So we can measure it by the average number of bits that we need to spend.

$$\frac{1}{n}\sum_{t=1}^{n} -\log p(x_t \mid x_{t-1}, \ldots, x_1) \tag{10.4.7}$$

This makes the performance on documents of different lengths comparable. For historical reasons, scientists in natural language processing prefer to use a quantity called *perplexity* rather than bitrate. In a nutshell, it is the exponential of the

[137] https://www.gutenberg.org/files/2600/2600-h/2600-h.htm
[138] https://en.wikipedia.org/wiki/The_Little_Prince