The Pragmatic Programmers

Programming Machine Learning

From Zero to Deep Learning

Paolo Perrotta
edited by Meghan Blanchette

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/pplearn/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

Andy

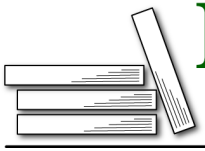
Programming Machine Learning

From Zero to Deep Learning

Paolo Perrotta

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-660-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: B8.0—August 21, 2019

Contents

Changes in the Beta Releases

Beta 8—August 21, 2019

- I added a new chapter about domesticating those wild deep networks: Chapter 18, Taming Deep Networks, on page 227.
- I also added a few more definitions in Appendix 2, The Words of Machine Learning, on page 269.

Beta 7—July 24, 2019

- Your final confrontation with overfitting awaits you in the new chapter: Chapter 17, Defeating Overfitting, on page 209.
- I made all the formulae nicer, bigger, and better-aligned.

Beta 6—July 10, 2019

- Let's start talking about deep learning! Check out the first chapter of Part III: Chapter 16, A Deeper Kind of Network, on page 195.
- I fixed a few errata and a few problems in the code throughout the book.

Beta 5—June 5, 2019

- I added the closing chapter of Part II: Chapter 15, Let's Do Development, on page 175.
- I added a few definitions to Appendix 2, The Words of Machine Learning, on page 269.
- I fixed figures across the book. Now they're more readable, especially on smaller screens and on black and white devices.
- I fixed more errata.

Beta 4—May 15, 2019

- I added a new chapter about testing: Chapter 14, The Zen of Testing, on page 169.

- The first chapter was missing an important concept. I added a new section to explain the fundamental idea of supervised learning. See The Math Behind the Magic, on page 8.
- I revised Chapter 2, Your First Learning Program, on page 15. Now it has more (and more readable) images, and cleaner explanations.

Beta 3—May 1, 2019

- I added a new chapter that covers some essential training techniques: Chapter 13, Batchin' Up, on page 157.
- I also added a new appendix that you can use as reference whenever you forget a term: Appendix 2, The Words of Machine Learning, on page 269. This appendix is still a work in progress, so if you search for a term that isn't there yet, feel free to suggest terms to add via the book's errata link.
- I fixed quite a few errata, unclear statements, and factual errors. Keep those errata coming!
- The high-level introduction in About This Book, on page ix needed a picture. Now it has it.

Beta 2—April 10, 2019

- We have a new chapter! I added Chapter 12, How Classifiers Work, on page 147.
- I fixed errata throughout. Thank you to everybody who reported them. (In case you didn't notice it, click the "Report erratum" link in the bottom of the PDF pages to report any errors, or go to the book's webpage at https://pragprog.com/book/pplearn/programming-machine-learning and click the errata link.)
- Those original "Hands On" boxes were sticking out. I rolled them into the main text.

Beta 1—March 20, 2019

- Initial beta release.

How the Heck Is That Possible?

Machine learning can seem like magic. How can a computer recognize the objects in an image? How can a car drive itself?

Those feats are baffling—not just to the layman, but to many software developers like you and me. Even after writing code for many years, I had no idea how machine learning could possibly work. While I tinkered with the latest web framework, someone out there was writing amazing software that looked like science fiction—and I couldn't even comprehend it.

I wanted in on the action. I wanted to be able to build those things myself.

I knew how to write software, so I assumed that I would grok machine learning quickly. I mean, how hard could it be? I put on a confident smile and started studying. Then I kept smiling confidently as I slammed my muzzle into a long sequence of brick walls.

To us developers, machine learning feels… *foreign*. The field is teeming with math jargon, researchy conventions and, frankly, bad code. Instead of tutorials, people point you at lectures and research papers. For many of us, machine learning is as intimidating as it's intriguing.

This is the book I missed when I got started with machine learning: an introduction for developers, written in our own language. After reading it, you'll be comfortable with the fundamentals, and able to write machine learning programs. No, you probably won't be able to build your own self-driving car just yet—but at least you will know how the heck that's possible.

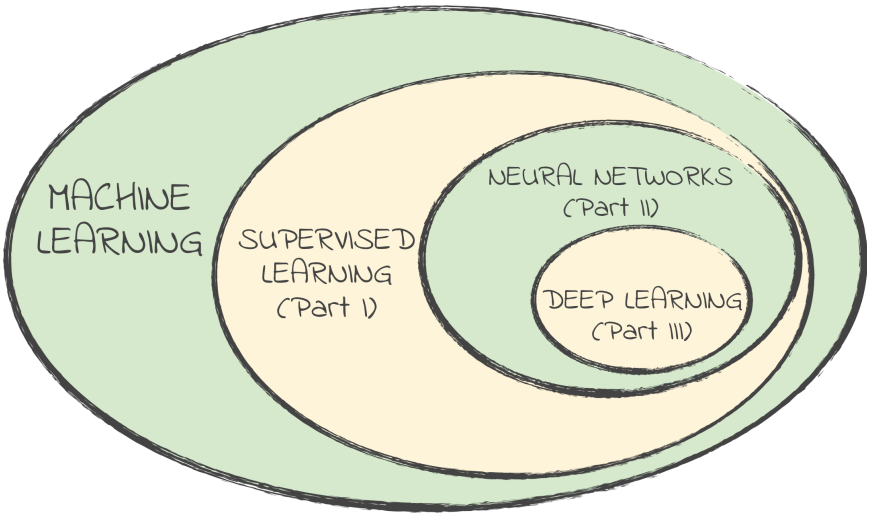
Come in.

About This Book

This is a book for developers who want to learn machine learning from scratch.

Machine learning is a broad field, and no book can cover it all. We'll focus on the three facets of machine learning that are most important today: *supervised*

learning, *neural networks*, and *deep learning*. We'll look into those terms as we go through the book, but here's a picture and a few quick definitions to get you started:



- *Supervised learning* is a specific type of machine learning. Machine learning comes in a few different flavors, but supervised learning is the most popular one these days. Part I of this book, *From Zero to Image Recognition*, is a hands-on supervised learning tutorial. Within a couple of chapters, we'll write a minimal learning program. Then we'll evolve the program step by step, until it's powerful enough to recognize handwritten characters. We'll craft this program all by ourselves, without using libraries. You'll understand each single line of code.

- There are many ways to implement a supervised learning system. The most popular of those is the *neural network*—a brilliant algorithm that was loosely inspired by the connections of neurons in our own brains. Part II of this book is dedicated to neural networks. We'll grow the program from Part I into a full-fledged neural network. We'll have to overcome a few challenges along the way, but the payoff will be worth it: the final neural network will be way more powerful than the starting program. Once again, we'll write the code ourselves, line by line. Its inner workings will be open for you to play with.

- Neural networks got a big boost in recent years, when researchers came up with breakthrough techniques to design and use them. This souped-up technology is vastly more powerful than the simple neural networks of old—so much so, that it got its own name: *deep learning*. That's also the title of Part III of this book. In it, we'll rewrite our neural network using

a modern machine learning library. The resulting code will be our starting point to understand what deep learning is about. Finally, as we wrap up the book, we'll take a look at a few advanced deep learning techniques, paving the way for your future explorations.

To be picky, things aren't quite as clean-cut as our picture implies. For example, neural networks can be used in other fields of machine learning, not just in supervised learning. However, the diagram above is a good starting point to get a sense of the topics in this book, and how they fit together.

Before We Begin

This book cannot turn you into a machine learning pro overnight, but it can give you an intuitive, practical understanding of how machine learning works. I want to open the hood of this discipline, show you the gears, and demystify the magic. Once you grasp the fundamental principles of machine learning, you'll find it much easier to dig deeper, incorporate these techniques in your daily job, and maybe even embark on a career as a machine learning engineer.

You don't need to be a senior developer to read this book. However, you should be comfortable writing short programs. If you know Python, then you just lucked out: that's the language that I will use throughout, so you'll feel right at home. Even if you don't know Python yet, no worries. It's a friendly language, and the code in this book will never get too complicated. Read Appendix 1, Just Enough Python, on page 251 to get up to speed, and be ready to Google for more information if you get stuck.

Machine learning involves a lot of mathematics. I won't dumb down the math, but I'll make it as intuitive as I can. You will need some high school math: I'm going to assume that you can read a Cartesian chart, that you know what the "axes" and their "origin" are, and that you can make sense of a function plot. Other than that, you don't need much math knowledge. In fact, you might be able to read through even if you consider yourself terrible at math… But be ready to be proven wrong about that!

Math Deep Dives

We all love intuitive math, but sometimes you might strive for a more formal explanation. If you ever feel lost while parsing a formula, or if you like mathematics and want to dig deeper, then look for "Math Deep Dive" boxes like this one. There are only a few of them—three or four in the entire book. They'll point you at relevant math screencasts on the excellent Khan Academy.[a] No matter what your current level of math is, this site has got you covered.

> Just to be clear, these additional lessons are optional. You don't need them to read this book—only if you wish to really wrap your mind around the mathematics of machine learning.
>
> a. www.khanacademy.org

On the other hand, if you have a solid background in linear algebra and calculus, then you might find some of the math obvious. In that case, feel free to breeze over the explanations you don't need.

Machine learning has a rich and specific vocabulary. You're likely to stumble upon new words, or new meanings for old words. Take it easy and don't feel like you have to remember everything. I will remind you of many of those words' meaning the next time we encounter them. Whenever a term gives you that obnoxious "I cannot quite remember what this means" feeling, you can look it up in Appendix 2, The Words of Machine Learning, on page 269.

One word about the datasets that I'll use in the examples: many of them are collections of images. Stay assured that machine learning can do much more besides image recognition: it can analyze text, generate music, or even hold conversations. However, image recognition makes for very intuitive examples, so it will be our go-to application throughout the book.

Finally, don't forget to visit this book's web page[1] and download the examples' source code. On the same page, you'll find a link to report errata—typos, bugs, and factual mistakes. The .pdf version of this book also contains a link to report errata in the bottom right corner of every page.

That's enough public service announcements. Let's dive into Part I.

1. https://pragprog.com/book/pplearn/programming-machine-learning

Part I

From Zero to Image Recognition

We're about to get an intro to supervised learning. Within a couple of chapters, we'll code our first machine learning system. Then we'll evolve this system, one small step at a time, until it becomes powerful enough to read handwritten digits.

You read that right: in the next 90 or so pages, we'll build an image recognition program. Even better, we're not going to use any machine learning library. Other than a few general-purpose functions for arithmetics and plotting, we'll write all of that code by ourselves.

In your future career, you're unlikely to ever write machine learning algorithms from scratch. But doing it once, to grok the fundamentals… that's priceless. You will understand every line in the final program. Machine learning will never look like magic again.

CHAPTER 1

How Machine Learning Works

Software developers like to share war stories. As soon as a few of us sit down in a pub, somebody asks: "What project are you working on?" Then we nod our heads off as we listen to each other's amusing, and sometimes horrible, tales.

In the mid 90s, during one of those evenings of bantering, a friend told me about the impossible mission she was on. Her managers wanted a program that would analyze X-ray scans and identify diseases, such as pneumonia.

My friend had warned management that the task was hopeless, but they refused to believe her. If a radiologist could do it, they reasoned, then why not a Visual Basic program? They even paired my friend with a professional radiologist, so that she could learn the job and turn it into code. That experience only reinforced her opinion that radiology required human judgment and intelligence.

We laughed at the futility of the task. A few months later, the project was canceled.

Fast-forward to more recent times. In late 2017, a research team at Stanford University published an algorithm to diagnose pneumonia from X-ray scans. The algorithm wasn't just OK—it was more accurate than professional radiologists. That was supposed to be impossible! How the heck could they write that code?

The answer is that they didn't. Instead of writing code, they cracked the problem with machine learning. Let's see what that means.

Programming vs. Machine Learning

Here's an example of the difference between machine learning (or simply "ML") and regular programming. Imagine building a program that plays video games. With traditional programming, that program might look something like this:

```
enemy = get_nearest_enemy()
if enemy.distance() < 100:
  decelerate()
  if enemy.is_shooting():
    raise_shield()
  else:
    if health() > 0.25:
      shoot()
    else:
      rotate_away_from(enemy)
else:
  # ...a lot more code
```

...and so on. Most of the code would be a big collection of `if..else` statements, mixed with imperative commands such as `shoot()`.

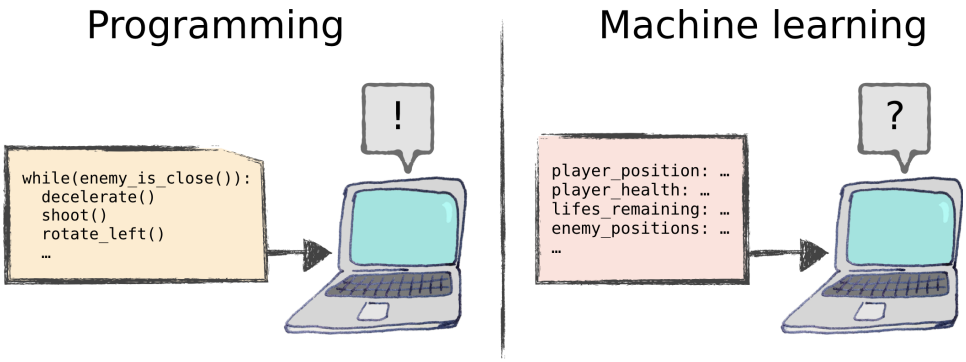
Granted, modern languages give us the means to replace those ugly nested `if`s with more pleasant constructs—polymorphism, pattern matching, or event-driven calls. The core idea of programming, however, stays the same: you tell the computer what to look for, and you tell it what to do. You must list every condition and define every action.

This approach has served us well, but it has a few flaws. First: you must be exhaustive. You can probably imagine dozens or hundreds of specific situations that you'd have to cover in that video game-playing program. What happens if the enemy is approaching, but there is a power-up between you and the enemy, and the power-up is shielding you from enemy fire? A human player would quickly notice the situation and take advantage of it. Your program... well, it depends. If you coded for that special case, then your program will deal with it—but we know how hard it is to cover all special cases, even in structured domains like accounting. Good luck listing each and every possible special case in complex domains like playing video games, driving a truck, or recognizing an image!

Even if you could list all those decisions, you'd have to know how to take them in the first place. That's a second limitation of programming, and a showstopper in some domains. For example, take a *computer vision* task like our original problem: identifying pneumonia in chest scans.

We don't *really* know how a human radiologist recognizes pneumonia. Yes, we have a high-level idea of it, like: "the radiologist looks for opaque areas". However, we don't know how the radiologist's brain recognizes and evaluates an opaque area. In some cases, the expert herself cannot tell you how she came to a diagnosis, except for a rather vague: "I know by experience that pneumonia doesn't look like this". Since we don't know how those decisions happen, we cannot instruct a computer to take them. That's a problem shared by all typically human tasks, such as tasting beer, or understanding a sentence.

Machine learning, on the other hand, turns traditional programming on its head: instead of giving *instructions* to the computer, ML is about giving *data* to the computer, and asking it to figure out what to do.



The idea of a computer "figuring out" anything sounds like wishful thinking, but there are actually a few different ways to make it happen. In case you're wondering, all of them still require running code. That code, however, isn't a step-by-step procedure to solve to the problem, like in traditional programming. Instead, the code in machine learning tells the computer how to crunch the data, so that the computer can solve the problem by itself.

As an example, here is one way that a computer can figure out how to play a video game. Imagine an algorithm that learns how to play by trial and error. It starts by giving random commands: "shoot," "decelerate," "rotate," and so on. If those commands eventually lead to success, such as a higher score, the algorithm remembers this experience. If they lead to failure, such as death, the algorithm also takes note. At the same time, it also takes note of the state of the game: where are the enemies, the obstacles, and the power-ups? How much health do we have? And so on.

From then on, whenever it encounters a similar game state, the algorithm is a bit more likely to attempt the successful actions than the unsuccessful

ones. After many cycles of trial and error, such a program would become a competent player. In 2013, a system using this approach reached superhuman skills in a bunch of old Atari games.[1]

This style of ML is called *reinforcement learning*. Reinforcement learning works pretty much like dog training: "good" behavior is rewarded, so that the dog does more of it.

(I also tried the same approach with my cat. So far, I failed.)

Reinforcement learning is just one way to let a computer figure out a problem. In this book, we'll focus on another style of machine learning—arguably the most popular one. Let's talk about it.

Supervised Learning

Amongst the various approaches to ML, *supervised learning* is the one that reaped the most impressive results so far. Here is how supervised learning solves a problem like diagnosing pneumonia.

To do supervised learning, we need to start from a set of *examples*, each carrying a *label* that the computer can learn from. For instance:

What are we building?	Each example could be…	Each label could be…
A system that identifies a dog's breed from its barking.	…a `.wav` recording of a dog.	…the dog's breed, like "greyhound" or "beagle".
A system that detects pneumonia.	…an X-ray scan.	…a boolean flag: 1 if the scan shows pneumonia, 0 if it doesn't.
A system that predicts the earnings of a lemonade stand from the weather.	…the recorded temperatures on a day in the past.	…the recorded earnings on the same day.

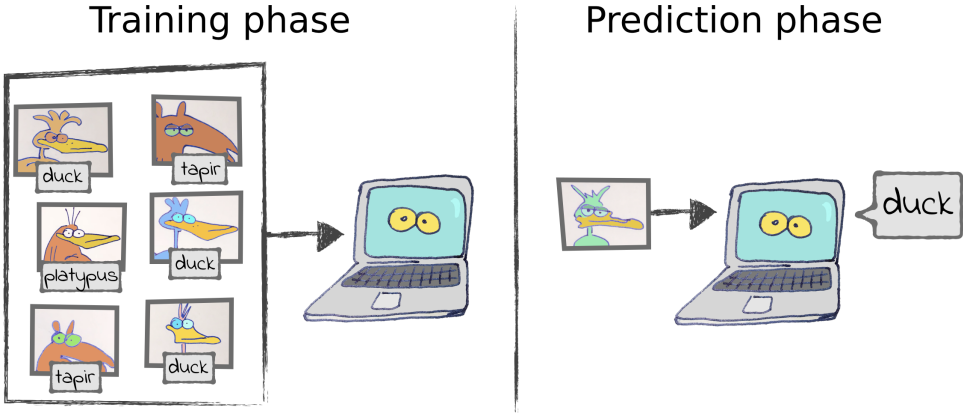
As you can see, examples can be a lot of different things: data, text, sound, video, and so on. Also, labels can be either *discrete* (as in the case of the dog breed detector) or *continuous* (as in the case of the temperature-to-lemonade converter). With some imagination, you can come up with many problems like the ones above, where you want to predict something from something else.

1. deepmind.com/research/publications/playing-atari-deep-reinforcement-learning

So, let's assume that we already put together a collection of labeled examples. Now we can dive into the two phases of supervised learning:

1. During the first phase, we feed the labeled examples to an algorithm that's designed to spot patterns. For example, the algorithm might notice that all pneumonia scans have certain common characteristics—maybe certain opaque areas—that are missing from non-pneumonia scans. This is called the *training phase*, because the algorithm is looking at the examples over and over, and learning to recognize those patterns.
2. Now that the algorithm knows what pneumonia look like, we switch to the *prediction phase*, where we reap the benefits of our work. We show an *unlabeled* X-ray scan to the trained algorithm, and the algorithm tells us whether it contains signs of pneumonia or not.

Here comes another example of supervised learning—a system that recognizes animals. Each input is the picture of an animal, and each label is the species. During the training phase, we show labeled images to the algorithm. During the prediction phase, we show it an unlabeled image, and the algorithm guesses the label:



Supervised learning is not limited to images. For example, you could have a system that recognizes spoken words. Such a system would use sound snippets as examples, and English words as labels. One more example: you could build a system that recognizes the sentiment behind a politician's tweet. In this case, the examples would be tweets, and the labels would be emotional states such as "angry", "aggressive", "absolutely furious", and so on.

Now that you've read this high-level explanation of supervised learning, you might have more questions than you started out with. We said that a supervised learning program "notices common characteristics" in the data and

"spots patterns"—but how? Let's step down one level of abstraction, and see how that magic happens.

The Math Behind the Magic

To understand the relation between a piece of data and its label, a supervised learning system exploits a mathematical concept—the idea of *approximating a function*. Let's see how that idea works, with a concrete example.

Imagine that you have a solar panel on your roof. You'd like a supervised learning system that learns how the solar panel generates energy, and predicts the amount of energy generated at some time in the future.

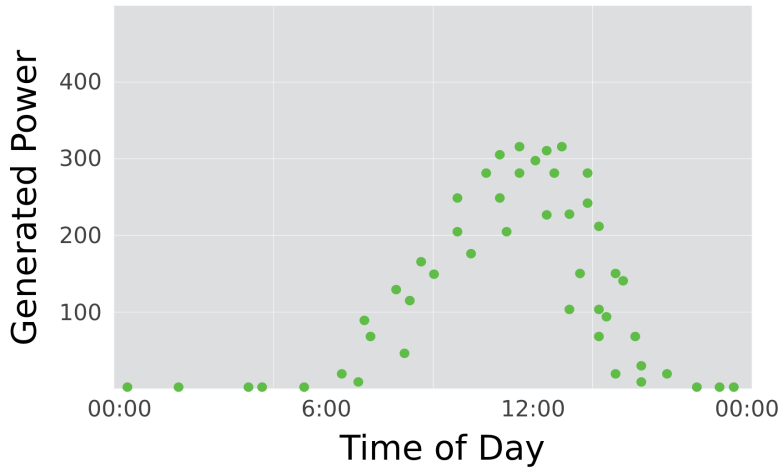
There are a few variables that impact the solar panel's output: the time of day, the weather, and so on. The time of day looks like an important variable, so you decide to focus on that one. In true supervised learning fashion, you start by collecting examples of power generated at different times of the day. After a few weeks of random sampling, you get a spreadsheet table that looks like this:

Time of day	Power (in Watts/hour)
09:01	153
11:48	280
05:20	0

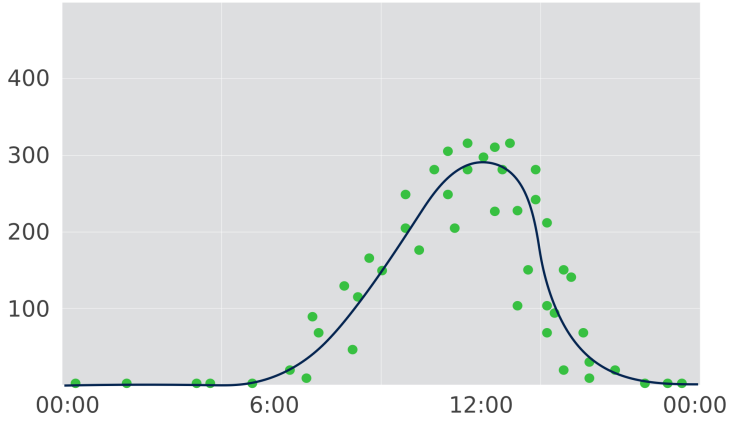
…and so on.

Each line in the table above is an example that includes an *input variable* (the time of day) and a label (the generated power)—just like in the system that recognizes animals, the picture is the input, and the name of the animal is the label.

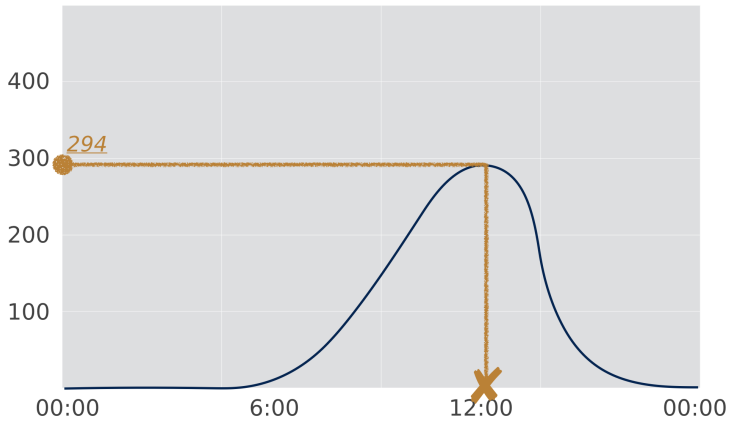
If you plot the examples, you can see visually how the time of day relates to the energy produced:



At a glance, our human brains can see that the solar panel doesn't generate power during the night, and that its power peaks around noon. Lacking the luxury of a brain, a supervised learning system can understand the data by approximating it with a function, like this:



Finding the function that approximates the examples is the hard part of the job. The prediction phase that follows is easier: the system forgets all about the examples, and uses the function to predict the power generated by the solar panel—for example, on any day at noon:



That's what I meant when I said that supervised learning works by approximating functions. The system receives real world data that's generally messy and incomplete. During the training phase, it approximates that complicated data with a relatively simple function. During the prediction phase, it uses that function to predict unknown data.

As a programmer, you're used to thinking of the many ways something could go wrong—so you might already be thinking of ways to complicate our example. For one, the output of a solar panel is influenced by other variables besides the time of day, like the cloud cover, or the time of the year. If we collected all those variables, we'd end up with a multi-dimensional cloud of points that we couldn't visualize on a simple chart. Also, in the case of the solar panel, we're predicting a *numerical* label. You might wonder how this method translated to non-numerical labels like the names of animals—also known as *categorical* labels.

We'll discuss all those matters in the book. For now, just know this: no matter how many complications you layer on top of it, the basic idea of supervised learning is the one we've just described: take a bunch of examples, and find a function that approximates them.

Modern supervised learning systems got very good at this approximation job. So good, in fact, that they can approximate extremely complicated relations—such as the one between an X-ray scan and a diagnosis. Granted, a function that approximates those relations would look maddeningly complicated to us humans. However, a computer program can find such a function, and use it to predict unknown data.

And that's the supervised learning in a nutshell. As for the details… well, the rest of this book is all about them. Let's set up our computers and start coding.

What About Unsupervised Learning?

There are a few different types of ML. Most tutorials list three of them in particular: reinforcement learning, supervised learning, and a third flavor called *unsupervised learning*. We used reinforcement learning to introduce the basics of ML, and we'll talk about supervised learning for the rest of this book. So, what about unsupervised learning?

Any practitioner will tell you that unsupervised learning is about learning from *unlabeled* data, just like supervised learning learns from labeled data. If you dig deeper, however, you'll realize that unsupervised learnings has little to do with supervised learning—or even with the intuitive idea of "learning". In fact, unsupervised learning looks more like a sophisticated form of data processing.

Here is an example of unsupervised learning: imagine that you're doing market research for an online shop. You have all the data for the shop's customers: how much they spent, how many times they visited, and so on. An unsupervised learning algorithm could help you make sense of that data, by grouping similar customers together—a process known as *clustering*. The machine doesn't know what the data means. It only "learns" how to cluster similar data into similar groups.

Unsupervised learning can be very useful, but it probably isn't what you think about when you think about ML. We won't talk about it in this book.

Setting Up Your System

We're about to embark on an ML tutorial that spans the entire first part of this book. We'll start from scratch and end with a working computer vision program. Later on, that program will be our starting point toward higher peaks: neural networks in Part II of this book, and deep learning in Part III.

You might want to approach this tutorial hands-on, running the source code for this book, and solving the exercises at the end of most chapters. Alternately, you might prefer to read through and get the big picture before you grab the keyboard. Both approaches are feasible, although programmers often go for the first.

If you prefer the hands-on approach, it won't take long to set up your system and run this book's code. Even though ML tends to require a lot of computing power, the code in this book runs fine on a regular laptop. You just need to install some software.

First and foremost, you need Python. It's the most popular language in the ML community, and the language that all of this book's examples are written in. Don't worry if you never coded in Python before—you'll be surprised by how readable this language is. If Python code perplexes you, then read

Appendix 1, Just Enough Python, on page 251. That will be enough knowledge to get you through this book.

A Note for Experienced Pythonistas

If you're steeped in Python, you might notice that the code in this book deviates from common Python conventions. For example, I actively avoid language-specific idioms such as list comprehension, that complicate the code for newcomers to the language. With the same intent of making the code more accessible, I might use slightly imprecise language, such as the word "function" in place of "method".

I apologize in advance for those transgressions to the Pythonic canon.

Let's get down to business and check that you have Python installed. Run:

```
python3 --version
```

If you don't have Python 3, then stop reading for a minute and go get it.[2] One thing to note: on some systems, you can also execute Python 3 by typing `python`, without the `3` at the end. On other systems, however, the `python` command executes Python 2. To avoid confusion and mysterious errors related to older Pythons, I'll always use the more explicit `python3` command in this book.

Now that you have the language, let's talk libraries. You'll need three of them to begin with. The big one is NumPy, a library for scientific computing. We'll also use two libraries to plot charts. Matplotlib is the de facto standard for chart-plotting in Python. Seaborn sits on top of Matplotlib, and focuses on making the charts more pretty.

There are two ways to install those libraries: you can use `pip`, Python's official package manager, or you can use Conda, a more sophisticated environment manager that is popular in the ML community. If you're curious, Installing Packages with Conda, on page 265 delves deeper into the differences between pip and Conda. If you're in doubt, then just use pip.

To install the libraries with pip, run these commands:

```
pip install numpy==1.15.2
pip install matplotlib==3.0.3
pip install seaborn==0.9.0
```

2. https://wiki.python.org/moin/BeginnersGuide/Download

…and you're done. If you'd rather use Conda, then look in the source code's root folder. The readme.txt file that you'll find in there contains all the necessary instructions.

Finally, you need some kind of coding environment. Many ML tutorials use a system called Jupyter Notebooks to edit and run code in the browser. You don't have to use Jupyter to run the examples in this book. Being a developer, you know how to write and run a program, so go ahead and use your favorite text editor or IDE. On the other hand, if you already know and like Jupyter, that's fine: look into the notebooks directory for a Jupyter version of the book's code.

Let's recap: you have Python, a few essential libraries, and your favorite editor. That's all you need to get started.

And now, let's build a program that learns.

Getting Used to ML

As a developer, you're used to learning at a rapid pace. With ML, however, you're entering a new field. I'm not going to lie: the next three or four chapters are going to be tough. As you read them, you might feel like an absolute beginner—an exciting, but sometimes frustrating, place to be.

I know that feeling, and I can tell you that it's worth getting through. I remember my excitement when, for the first time, I ran a machine learning program that came up with accurate predictions. Hold on, and soon enough you'll know that geeky joy.

CHAPTER 2

Your First Learning Program

Welcome to our first stepping stone in machine learning. In this chapter, we're going to build a tiny supervised learning program. This program will be a long haul from our goal of image recognition—in fact, it won't have anything to do with computer vision at all. However, we'll improve its code over the next chapters, until it gets sophisticated enough to tackle images.

We'll base the first version of our program on a technique called *linear regression*. Remember what I said in the previous chapter? Supervised learning is about approximating data with a function. In linear regression, that function is as simple as it could be: a straight line.

However simple linear regression is, don't underestimate the challenge ahead. To introduce linear regression, this chapter will touch on many different concepts—so many, that you might feel slightly overwhelmed. Take it slow and easy. Even if it looks like we're just writing a short program, we're actually laying the foundation of our ML knowledge. What you learn here will stay useful throughout this book, and beyond.

Let's start with a practical problem.

Getting to Know the Problem

Our friend Roberto owns a cozy little pizzeria in Florence. Every day at noon, he checks the number of reserved seats and decides how much pizza dough to prepare for dinner. Too much dough, and it goes wasted; too little, and he runs out of pizzas. In either case, he loses money.

It's not always easy to gauge the number of pizzas from the reservations. Many customers don't reserve a table, or they eat something other than pizza. Roberto knows that there is some kind of link between those numbers, in

that more reservations generally mean more pizzas—but other than that, he's not sure what the exact relation is.

Roberto dreams of a program that looks at historical data, grasps the relation between reserved seats and pizzas, and uses it to forecast tonight's pizza sales from today's reservations. Can we please code such a program for him?

Supervised Pizza

Remember what I said back in Supervised Learning, on page 6? We can solve Roberto's pizza forecasting problem by training a supervised learning algorithm with a bunch of labeled examples. To get the examples, we asked Roberto to jot down a few days' worth of reservations and pizzas, and collected that data in a file. Here's what the first few lines of that file look like:

02_first/pizza.txt

```
Reservations  Pizzas
13            33
2             16
14            32
23            51
```

Each line is an example, composed of an input variable (the reservations) and a label (the pizzas). Once we have an algorithm, we can use these examples to train it. Later on, during the prediction phase, we can pass a specific number of reservations to the algorithm, and ask it to come up with a matching number of pizzas.

Let's start with the numbers, like a data scientist would.

Making Sense of the Data

If we glance at Roberto's examples, it seems that the reservations and pizzas are correlated. Let's fire up a Python shell (with the `python3` command) and take a deeper look.

The NumPy library has a convenient function to import whitespace-separated data from text:

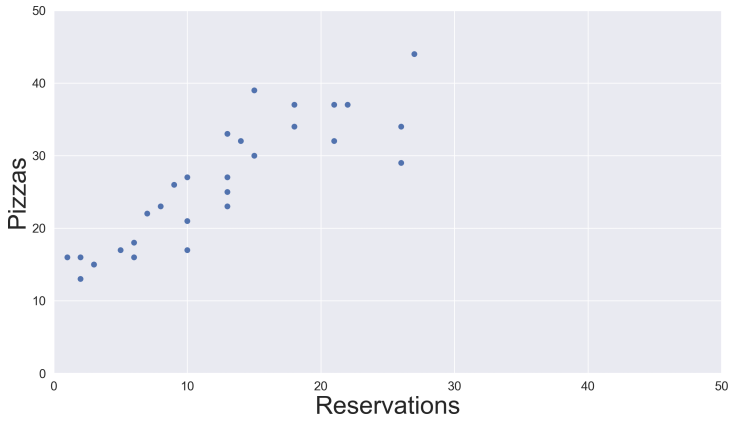
```
import numpy as np
X, Y = np.loadtxt("pizza.txt", skiprows=1, unpack=True)
```

I skipped the headers row, and I "unpacked" the two columns into separate arrays called `X` and `Y`. `X` contains the values of the input variable, and `Y` contains the labels. I used uppercase names for `X` and `Y`, because that's a common convention to indicate constants in Python.

Let's peek at the data, to make sure it loaded OK. If you wish to follow along, start a Python interpreter, send the two lines above, and then print the first few elements of `X` and `Y`:

```
⇛ print(X[0:5], Y[0:5])
❮ [ 13.   2.  14.  23.  13.] [ 33.  16.  32.  51.  27.]
```

The numbers are consistent with Roberto's file, but it's still hard to make sense of them. Plot them on a chart, though, and they become clear:



Now the correlation jumps out at us: the more reservations, the more pizzas. To be fair, a statistician might scold us for drawing conclusions from a handful of hastily collected examples. However, this is no research project—so let's ignore the little statistician on our shoulder, and build a pizza forecaster.

The Plotting Code

In case you're curious, here's the code that generates the plot in Making Sense of the Data, on page 16:

02_first/plot.py

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

sns.set()                                                  # activate Seaborn
plt.axis([0, 50, 0, 50])                                   # scale axes (0 to 50)
plt.xticks(fontsize=15)                                    # set x axis ticks
plt.yticks(fontsize=15)                                    # set y axis ticks
plt.xlabel("Reservations", fontsize=30)                    # set x axis label
plt.ylabel("Pizzas", fontsize=30)                          # set y axis label
X, Y = np.loadtxt("pizza.txt", skiprows=1, unpack=True)    # load data
plt.plot(X, Y, "bo")                                       # plot data
plt.show()                                                 # display chart
```

Besides NumPy, the code above uses two libraries: Matplotlib draws charts, and Seaborn makes them look pretty. The `plot()` function plots the examples as blue circles

(that's what 'bo' stands for), while the rest of the code sets up the axes, loads the code, and displays the chart. If you wish, you can run this program yourself with `python3 plot.py`.

You don't need to understand the plotting code above, but sooner or later you'll probably write similar code yourself. Maybe set aside a rainy Sunday to learn Matplotlib and Seaborn in the future—or just do it bit by bit, as you need them.

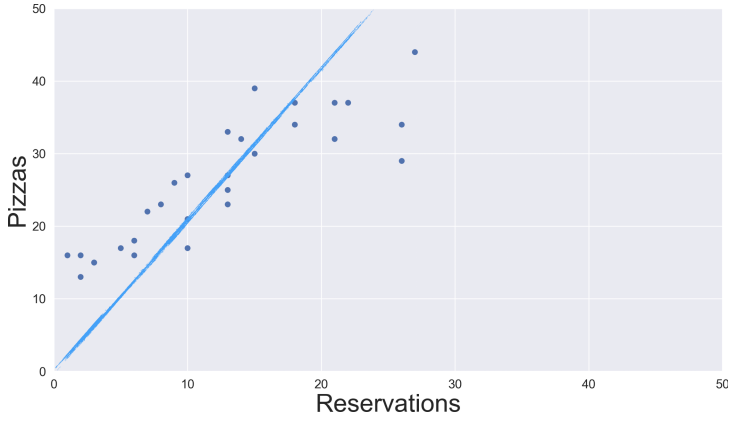
In the rest of this book, I'll skip the code that generates the plots. You can always find it in the book's accompanying source code.

Coding Linear Regression

To recap our goal: we want to write a program that calculates the number of pizzas from the number of reservations. That program should follow the approach we discussed in The Math Behind the Magic, on page 8: during the training phase, the program approximates the data with a function; then, during the prediction phase, it uses the function to infer the number of pizzas.

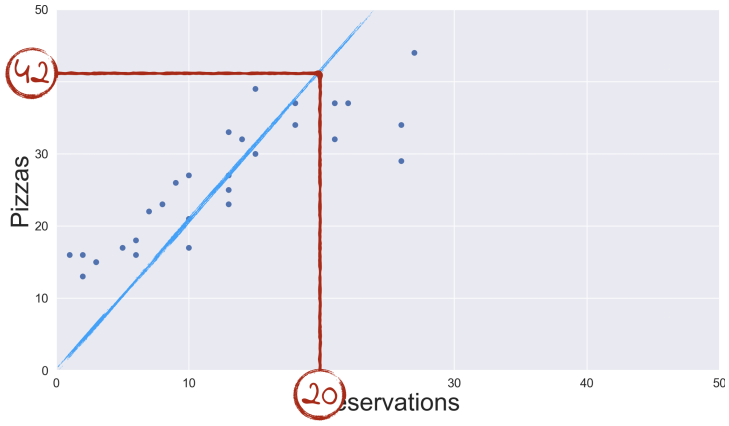
In the general case, finding a function that approximates the data can be a lot of work. In our specific case, however, we lucked out. Our data points are roughly aligned, so we can approximate the data with an especially simple function: a line.

Let's see what that line would look like. For now, let's pick a line that passes by the origin of the axes, because that will make things easier to begin with.



Once we've found the line, the training phase is over. You can say that the line is our *model* of the relation between reservations and pizzas.

Now we can move on to the prediction phase, where we use the line to predict the label from the input variable. Here is an example: we have 20 reservations; how much pizza do we expect to sell? To answer that question, I picked the point x = 20 on the "Reservations" axis. From there I traced up until I crossed the line, and then I traced left until I crossed the "Pizzas" axis. I ended up at y = 42.



So, there you have it: with 20 reservations, we can expect to sell about 42 pizzas.

This method has been used by statisticians since way before supervised learning existed. It's called *linear regression*. "Linear" means that we're tracing a straight line rather than a curve, and "regression" is a statistician's way of saying: "find the relation between two variables".

To recap, here is how you do supervised learning with linear regression:

1. *Training phase:* trace a line that approximates the examples;
2. *Prediction phase:* use the line to predict the label from the input variable.

Admittedly, not every relation can be approximated well with a straight line. If your examples fall along a curve, or lack a discernible shape, then you can't get away with this simple method. However, Roberto's examples roughly fall on a line, so linear regression should be good enough for this particular problem.

Now let's find a way to implement linear regression with code.

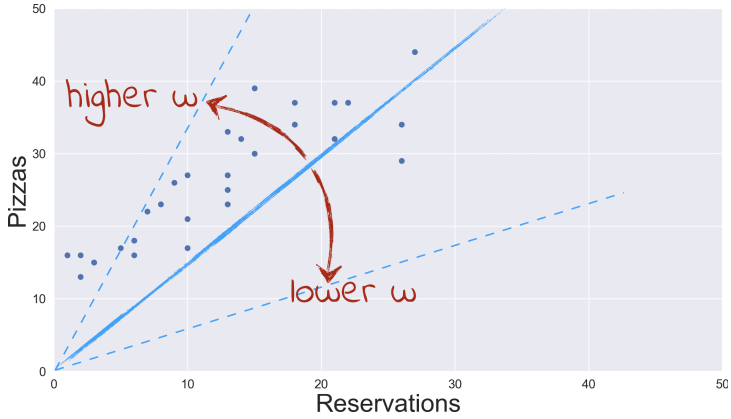
Defining the Model

To turn linear regression into running code, we need some way to represent the line numerically. That's where mathematics comes into the picture.

Here is the mathematical equation of a line that passes by the origin of the axes:

```
y = x * w
```

You might remember this equation from your studies, but don't fret if you don't. Here is what it means: each line passing by the origin is uniquely identified by a single value. I called this value `w`, short for *weight*. You can also think of `w` as the *slope* of the line: the higher `w` is, the steeper the line.



Math Deep Dive: Linear Equations

If the equation `y = x * w` confuses you, then you might want to watch Khan Academy's screencasts on linear equations.[a] These optional videos offer much more content than you need to read this book—so don't feel like you have to watch them all.

a. www.khanacademy.org/math/algebra/two-var-linear-equations

Before we move on, let me rewrite the equation of the line with a slightly different notation:

```
ŷ = x * w
```

I changed the symbol `y` to $\hat{y}$ (read "y-hat"), because I don't want to confuse this value with the `y` values that we loaded from Roberto's file. Both symbols represent the number of pizzas, but there is a crucial difference between the two. $\hat{y}$ is a forecast—our prediction of how many pizzas we hope to sell. By contrast, the labels `y` are real-life observations.

After this short digression about notation, let me go back to the important concept here: `w` is a constant that identifies the line. In other words, `w` is all we need to represent a line in code.

So, let's write that code. We'll start from the prediction phase, because it's easier than the training phase. We will tackle the training phase soon afterwards.

Implementing Prediction

Imagine that somebody came to you with a line (that is, a value of `w`), and asked you to use that line to predict the value of $\hat{y}$ from `x`—like, pizzas from reservations. That's a one-liner:

02_first/linear_regression.py

```
def predict(X, w):
    return X * w
```

The `predict()` function predicts the pizzas from the reservations. To be more precise, it takes the input variable and the weight, and it uses them to calculate $\hat{y}$.

This tiny function is more powerful than you may think. In particular, `X` can be either a single number, or an entire array of reservations. NumPy comes with its own array type, that supports *broadcast* operations: if we multiply an array of reservations by `w`, then NumPy multiplies each element of the array by `w`, and returns an array of predicted pizzas. That's a handy way of making multiple predictions at once.

In the beginning of Coding Linear Regression, on page 18, we used a hand-drawn line to match reservations to pizzas. The `predict()` function does the same thing that we did by hand, but it's more precise. How many pizzas do we expect to sell if we have 20 reservations? Let's say that our line is `w = 2.1`. Call `predict(20, 2.1)`, and you get back the predicted number of pizzas: 42.

That's all we need for the second phase of linear regression. Now let's take care of the more complicated first phase. It will take us four intense pages to do that, so hold on tight.

Implementing Training

Now we want to write code that implements the first part of linear regression: given a bunch of examples (`X` and `Y`), it finds a line `w` that approximates them. Can you think of a way to do that? Feel free to stop reading for a minute and think about it. It's a fun problem to solve.

You might think that there is one simple way to find `w`: just use math. After all, there must be some formula that takes a list of points and comes up with a line that approximates them. We could Google for that formula, and maybe even find a library that implements it.

As it turns out, such a formula does indeed exist… but we won't use it, because that would be a dead end. If we use a formula to approximate these points with a line, then we'll get stuck later, when we tackle datasets that require twisty functions. We'd better look for a more generic solution—one that works for any dataset.

So much for the mathematician's approach, then. Let's look at a programmer's approach instead.

How Wrong Are We?

Here is one strategy to find the best line that approximates the examples. Imagine if we had a function that takes the examples (`X` and `Y`) and a line (`w`), and measures the line's error. The better the line approximates the examples, the lower the error. If we had such a function, we could use it to evaluate multiple lines, until we find a line with a low enough error.

Except that instead of "error", the cool ML kids have another name for this function: they call it the *loss*.

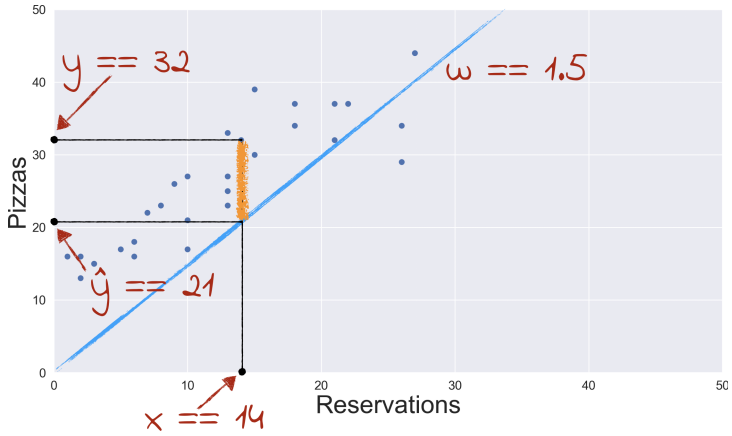
Here is how we can write a loss function. Assume that we've come up with a random value of `w`—say, 1.5. Let' use this `w` to predict how many pizzas we'd sell if we had, say, 14 reservations. Call `predict(14, 1.5)`, and you get $\hat{y} = 21$ pizzas.

But here is a crucial point: we *know* from our examples that 21 is the wrong value. Look back at the first few examples:

02_first/pizza.txt

```
Reservations  Pizzas
13            33
2             16
14            32
23            51
```

Look at that day with 14 reservations. On that night Roberto sold 32 pizzas, not 21. So we can calculate an error that is the difference between the predicted value $\hat{y}$ and the examples—that thick orange segment below:



Here is what that calculation would look like in code:

```
error = predict(x, w) - y
```

There's a wrinkle in the calculation above: `error` could be zero, positive, or negative. However, an error should always be positive. If you add multiple errors together, which we're going to do soon, you don't want two opposite-sign wrongs to make one right. To guarantee that the error is always positive, let's square it:

```
squared_error = error ** 2
```

We could also use the absolute value of the error instead of squaring it. However, squaring the error has additional benefits that will become obvious in the next chapter.

Now let's average the squared errors of all the examples, and *voilà*! We finally have our loss. This way to calculate the loss is called the *mean squared error*, and it's pretty popular amongst statisticians. Here's what it looks like in code:

02_first/linear_regression.py

```
def loss(X, Y, w):
    return np.average((predict(X, w) - Y) ** 2)
```

Remember that we used NumPy to load `X` and `Y`? Both variables are NumPy arrays, which makes for pretty terse code. In the span of the first line of `loss()`, we multiply each element of `X` by `w`, resulting in an array of predictions; for each prediction, we compute the error—the difference between the prediction and the matching label; we square each error with the ** power operator; and finally, we ask NumPy to average the squared errors. And there you have it: our mean squared error.

Now that we're done with the loss() function, we can write the last function of our learning program.

Lingo Overload

"Mean squared error", "loss", "weight"... In these first few pages of the book, new names are coming up fast and hard. Here's a friendly reminder: the meaning of the most important terms that we're using is in Appendix 2, The Words of Machine Learning, on page 269. If you don't find a specific term in that appendix, then look it up in the book's index.

Closer and Closer

Remember what the training phase of linear regression is about: we want to find a line that approximates the examples. In other words, we want to calculate w from the measured values in X and Y. We can use an iterative algorithm to do that:

02_first/linear_regression.py

```
def train(X, Y, iterations, lr):
    w = 0
    for i in range(iterations):
        current_loss = loss(X, Y, w)
        print("Iteration %4d => Loss: %.6f" % (i, current_loss))

        if loss(X, Y, w + lr) < current_loss:
            w += lr
        elif loss(X, Y, w - lr) < current_loss:
            w -= lr
        else:
            return w

    raise Exception("Couldn't converge within %d iterations" % iterations)
```

The train() function goes over the same examples over and over, until it learns how to approximate them. It takes the examples, a number of iterations, and another argument called lr (which I will explain in a moment). The algorithm starts by initializing w to an arbitrary value of 0. This w represents a line on the chart. It's unlikely to be a good approximation of the examples, but it's a start.

Afterwards, train() gets into a loop. Each iteration starts by calculating the current loss. Then it considers an alternative line—the one that you get when you increase w by a small amount. That amount is usually called lr, which stands for *learning rate*.

We just added the learning rate to w, which results in a new line. Does this new line result in a lower loss than our current line? If so, then w + lr becomes the new current w, and the loop continues. Otherwise, the algorithm tries another line: w - lr. Once again, if that line results in a lower loss than the current w, the code updates w and continues the loop.

If neither w + lr nor w - lr yield a better loss than the current w, then we're done. We've approximated the examples as well as we can, and we return w to the caller.

To put it in more concrete terms, this algorithm is rotating the line up and down, making it a tad steeper or a tad less steep at each iteration, while keeping an eye on the loss. The higher the learning rate, the faster the system moves the line. Picture a radio operator of old, slowly turning a knob to make the voice in his headset that tiny little bit clearer—until, eventually, it's as clear as it gets.

Iterative algorithms can sometimes get stuck in an infinite loop. (They don't *converge*). A computer scientist could prove that this particular algorithm doesn't have that problem: given enough time and iterations, it will always converge. However, the algorithm might run out of iterations before that happens. In that case, train() gives up and exits with an exception.

I know that you're itching to run this code. Let's do it.

Let's Do This!

It's time to write and run our "main" function. The code below feeds Roberto's examples to train(). It uses Python's named arguments to be more explicit, and it prints out the final value of the weight, plus the predicted pizzas for 20 reservations:

02_first/linear_regression.py

```
if __name__ == "__main__":
    # Import data
    X, Y = np.loadtxt("pizza.txt", skiprows=1, unpack=True)

    # Train system
    w = train(X, Y, iterations=10000, lr=0.01)
    print("\nw=%.3f" % w)
    print("Prediction: x=%d => y=%.2f" % (20, predict(20, w)))
```

Python's __main__

The code in Let's Do This!, on page 25 opens with a common Python idiom:

```
if __name__ == "__main__":
    # some code
```

In a nutshell, this idiom means: "only execute the following code if someone runs the file as a program", for example by typing python3 linear_regression.py. On the other hand, if you want to use functions such as train() in your own code, then you can import linear_regression.py from another program, with the instruction import linear_regression. In that case, this "main" code is not executed.

You can read more about Python's _main_ in The main Idiom, on page 263.

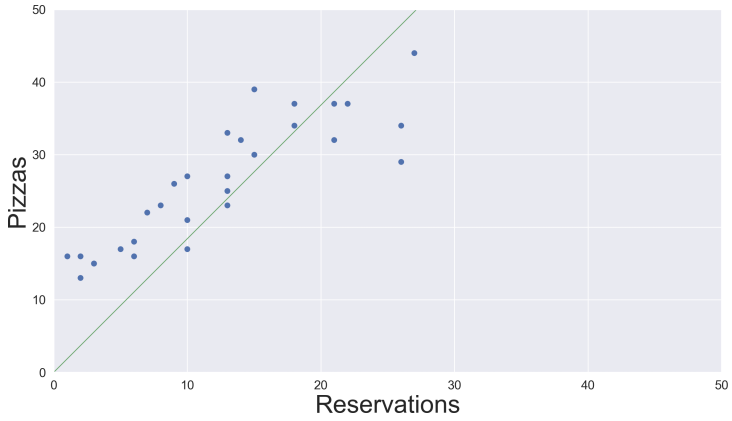
When we call train(), we need to pick values for iterations and lr. For now, we can do that by trial and error. I asked for 10000 iterations, which feels like a good value to begin with. As for lr, remember what it does: it decides how much w changes at each step of training. I set it to 0.01, which seems precise enough for the task at hand. ("Enumerating pizzas", as opposed to, for example, "weighing electrons").

Indeed, running the program shows that train() converges on a result in less than 200 iterations:

```
Iteration    0 => Loss: 812.866667
Iteration    1 => Loss: 804.820547
Iteration    2 => Loss: 796.818187
…
Iteration  184 => Loss: 69.123947

w=1.840
Prediction: x=20 => y=36.80
```

Success! The loss decreased at each iteration, until the algorithm gave up at squeezing it further. The weight at that point is 1.84, so that's the number of pizzas that Roberto can expect to sell for each reservation. In the case of 20 reservations, he can expect to sell around 36.80 pizzas. (Pizzas are not really sold in fractions, but we like to be precise).

By computing w, our code did the equivalent of "drawing a line on the chart". Let's visualize that line. (As usual, you'll find the plotting code in the book's source code.)



Nice. But we can make it even better.

Adding a Bias

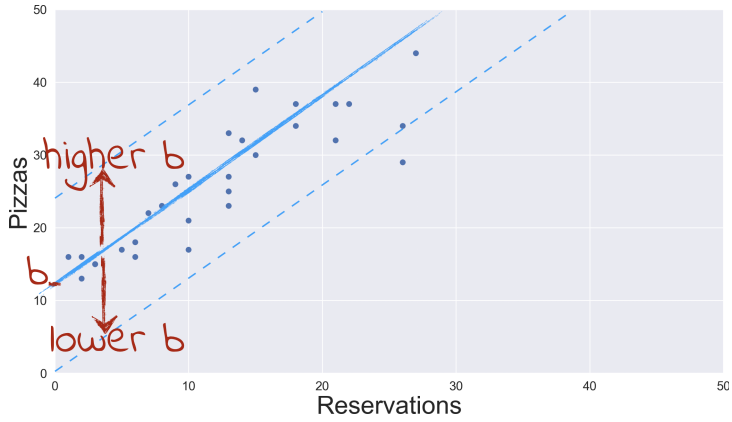
Look at the chart we just plotted. Our line is not quite the best approximation of the examples, is it? The perfect line would have less slope, and it wouldn't pass by the origin. Give or take, it would cross the "Pizzas" axis around the value of 10.

So far, we forced the line to pass by the origin to keep our model as simple as we could. It's time to remove that constraint. To draw a line that is *not* constrained to pass by the origin, we need one more parameter in our model:

```
ŷ = x * w + b
```

You might experience *déjà vu* here. The equation above is the classic linear function which you may have studied in 8th grade. Most people would remember it as `y = m * x + b`, where `m` is called the "slope" and `b` is called the "y-intercept". Here, we'll use the vocabulary of machine learning instead. We already called `w` the "weight", and we will call `b` the *bias*.

Intuitively, the bias measures the "shift" of the line up or down the chart. If the bias is 0, then we're back at our previous case, with a line passing by the origin. If it's not, then the line crosses the `y` axis at a value equal to `b`.



Here is the entire linear regression program, updated to use the new model with two parameters. The small arrows in the left margin mark changes:

02_first/linear_regression_with_bias.py

```
import numpy as np

➤ def predict(X, w, b):
➤     return X * w + b

➤ def loss(X, Y, w, b):
➤     return np.average((predict(X, w, b) - Y) ** 2)

  def train(X, Y, iterations, lr):
➤     w = b = 0
      for i in range(iterations):
➤         current_loss = loss(X, Y, w, b)
          print("Iteration %4d => Loss: %.6f" % (i, current_loss))

➤         if loss(X, Y, w + lr, b) < current_loss:
              w += lr
➤         elif loss(X, Y, w - lr, b) < current_loss:
              w -= lr
➤         elif loss(X, Y, w, b + lr) < current_loss:
➤             b += lr
➤         elif loss(X, Y, w, b - lr) < current_loss:
➤             b -= lr
          else:
➤             return w, b

      raise Exception("Couldn't converge within %d iterations" % iterations)

  if __name__ == "__main__":
      # Import data
      X, Y = np.loadtxt("pizza.txt", skiprows=1, unpack=True)

      # Train system
➤     w, b = train(X, Y, iterations=10000, lr=0.01)
```

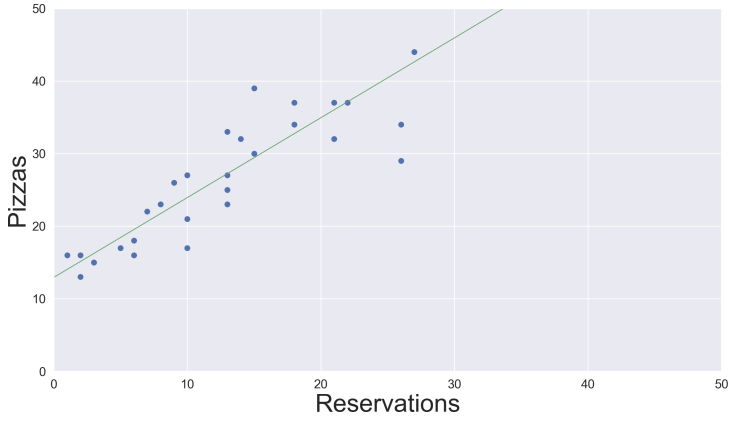
```
➤      print("\nw=%.3f, b=%.3f" % (w, b))
➤      print("Prediction: x=%d => y=%.2f" % (20, predict(20, w, b)))
```

Most lines have changed, but all of those changes are about introducing `b`. The most notable changes are in predict(), that uses the new model, and train(), that tweaks `b` the same way that it tweaks `w`, increasing or decreasing it while keeping an eye on the loss. I wasn't sure how to tweak `w` and `b` at the same time, so I went for one of those temporary hacks that we all love: I just added a couple of new branches to the if.

Let's give this program a shot. The train() function takes longer to converge this time, but eventually it does:

```
Iteration    0 => Loss: 812.867
Iteration    1 => Loss: 804.821
Iteration    2 => Loss: 796.818
…
Iteration 1551 -> Loss: 22.864

w=1.100, b=12.930
Prediction: x=20 => y=34.93
```

And here is what we get if we plot the data and the line:



Now we're really talking! The line approximates the examples much better, and it's crossing the `y` axis at a value equal to `b`. Even better, our final loss is lower than it used to, which means that we're more accurate than ever at forecasting pizza sales. Roberto will be delighted!

That was a lot of work for a first version of our program. Let's catch our breath, take a step back, and take a look at the bigger picture.

"Hyperparameters"?

In supervised learning, the training phase returns a set of values that we can use in the model—in our case, a weight and a bias that we can use in the equation of a line. Those values are called "parameters". That's unfortunate for us programmers, because we already use the name "parameter" for something different. It's easy to get confused between parameters such as `w` and `b`, and the parameters of a function such as `train()`.

To avoid the confusion, ML practitioners use another name for the parameters of the `train()` function: they call them *hyperparameters*, meaning "higher level parameters". To recap: we set *hyperparameters* such as `iterations` and `lr`, so that the `train()` function can find *parameters* such as `w` and `b`.

What You Just Learned

We went through a lot of information in these first two chapters, and I introduced many new terms. Let's recap.

In this chapter we wrote our first *supervised learning* program. A supervised learning system learns from *examples*, each composed of an *input* and a *label*. In our case, the inputs were the number of reservations, and the labels were the number of pizzas.

Supervised learning works by approximating the examples with a function, often called the *model*. In our first program, our model is a line identified by two *parameters*—the *weight* and the *bias*. This idea of approximating the examples with a line is called *linear regression*.

The first phase of supervised learning is the *training phase*, when the system tweaks the parameters of the model to approximate the examples. During this search, the system is guided by a *loss function* that measures the distance between the examples and the current approximation: the lower the loss, the better the approximation. Our program calculates the loss with a formula called the *mean squared error*. The result of training are a weight and a bias—the ones that result in the lowest loss that the system could find.

The parameters found during the training phase are then used during the second phase of supervised learning: the *prediction phase*. This phase passes an unlabeled input through the parametrized model. The result is a forecast, such as: "Tonight, you can expect to sell 42 pizzas."

You can find an analogy between the training and prediction phases of supervised learning, and the *compilation* and *runtime* phases of programming. Training tends to be hungry for data and require a lot of computation, while prediction tends to be cheap. Even in our little program, the `train()` function

takes a discernible time to find a line that fits the examples, while predict() is a blazing fast multiplication.

The differences between training and prediction become even more obvious for large systems: training a speech recognition system might involve weeks of number crunching through millions of audio files on multiple GPUs. Then you could conceivably deploy the system to a smartphone, and use it to cheaply predict the meanings of individual samples.

Phew! That was a lot of knowledge. I promise that no other chapter will introduce so many new terms at once.

On the plus side, in this chapter we've coded a supervised learning program from scratch, and that's quite an achievement. Most ML software today, including most of those amazing deep learning systems, uses supervised learning. Granted, those systems are way more complicated than our pizza forecaster. Rather than a list of reservations, they might eat high-resolution images for breakfast; and rather than a simple model with two parameters, they might have a complex model with tens of thousands of parameters. Still, they work on the same basic premises of our tiny Python program.

In the next chapter, we'll build on those premises, getting familiar with one of the most important algorithms in machine learning.

Hands On: Code Play

Before you move on, you might want to play with the code for a little while. That's optional, but it's a good way to make these concepts stick.

To begin with, you can start getting familiar with the system's hyperparameters. (If you don't know what a "hyperparameter" is, read "Hyperparameters"?, on page 30). Try changing the value of the lr argument to train(). What happens if you set lr to a very small value? What if you set it to a large value? What are we gaining and losing in the two cases?

CHAPTER 3

Walking the Gradient

In the previous chapter, we achieved something that we can be proud of: we wrote a piece of code that learns. If we got that code reviewed by computer scientists, however, they would find it somewhat lacking. In particular, the stern computer scientist would raise an eyebrow at the sight of the `train()` function. "This code might work OK for this simple problem," he or she would say, "but it won't scale to real-world problems."

Fair enough. In this chapter, we're going to address those concerns in two ways. First, we're *not* going to get our code reviewed by a computer scientist. Second, we're going to describe the problems with the current `train()` implementation, and fix them with one of machine learning's key ideas: an algorithm called *gradient descent*. Gradient descent isn't just useful for our tiny program. In fact, you cannot go very far in ML without gradient descent. In different forms, this algorithm will accompany us all the way to deep learning—so it's worth learning it now.

Let's start with the problem that gradient descent is meant to solve.

Our Algorithm Doesn't Cut It

Our program can successfully forecast pizza sales, but why stop there? Maybe we could use the same code to forecast other things, such as the stock market. We could get rich overnight! (Disclaimer: that wouldn't really work.)

If we tried to apply our current code to a different problem, however, we'd bump into an impediment. That code is based on a very simple model with two parameters: the weight `w` and the bias `b`. Most real-life problems require more complex models that have more than just two parameters. As an example, remember our goal for Part I of this book: we want to build a system that recognizes images. An image is way more complicated than a single

number, so we can expect that system to have many more parameters than the pizza forecaster.

Unfortunately, if we added more parameters to our model, we'd kill its performance. To see why, let's review the train() function from the previous chapter:

```
def train(X, Y, iterations, lr):
    w = b = 0
    for i in range(iterations):
        current_loss = loss(X, Y, w, b)
        print("Iteration %4d => Loss: %.6f" % (i, current_loss))

        if loss(X, Y, w + lr, b) < current_loss:
            w += lr
        elif loss(X, Y, w - lr, b) < current_loss:
            w -= lr
        elif loss(X, Y, w, b + lr) < current_loss:
            b += lr
        elif loss(X, Y, w, b - lr) < current_loss:
            b -= lr
        else:
            return w, b

    raise Exception("Couldn't converge within %d iterations" % iterations)
```

Remember how train() works? At each iteration, it tweaks either w or b, looking for the values that minimize the loss. As I hinted in the last chapter, however, this algorithm is unreliable. Here is one way in which it could go wrong: as we tweak w, we might be increasing the loss caused by b, and the other way around. To avoid that problem and get as close as possible to the minimum loss, we should tweak both parameters *at once*. The more parameters we have, the more important it is to tweak them all at the same time.

To tweak w and b together, we'd have to try all the possible combinations of tweaks: increase w and b; increase w and decrease b; increase w and leave b unchanged; decrease w and… well, you got my point. Do the math, and you'll find that the total number of tweaking combinations would be 3 to the power of the number of parameters. With two parameters, that would be 3^2—that is, 9 combinations.

Nine calls per iteration don't sound like a big deal—but increase the number of parameters to 10, and you get 3^{10} combinations, which is almost 60,000 calls to loss() per iteration. You might think that 10 parameters are far-fetched, but they aren't. Later in this book we're going to fit *hundreds of thousands* of parameters to *megabytes* of data. With those problems, an algorithm that tries every combination is never going to fly. We should nip this slow code in the bud.

There is also a more urgent problem in the current implementation of train(): it tweaks parameters in increments that are equal to the learning rate. If lr is large, then the parameters change quickly, which speeds up training—but the final result is less precise, because each parameter has to be a multiple of that large lr. To increase precision, we need a small lr, which results in even slower training. We're trading off speed for precision, when we actually need both.

That's why our current code is basically a hack. We should replace it with a better algorithm—one that makes train() both fast *and* precise.

Gradient Descent

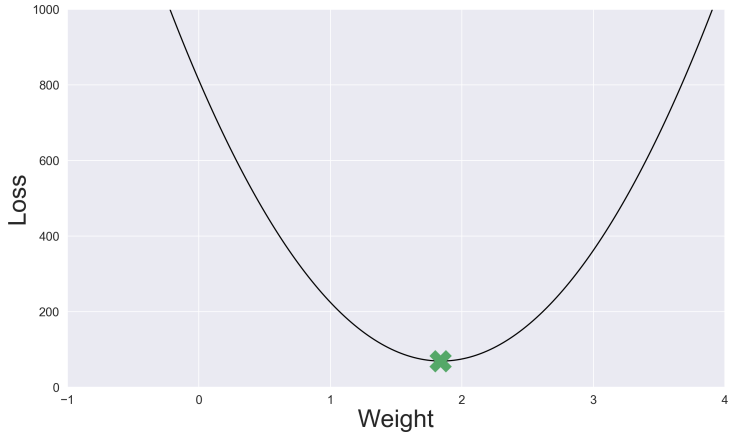
It's time to look for a better train() algorithm. In this section, we're going to do that.

The job of train() is to find the parameters that minimize the loss. Let's focus on loss() itself:

```
def loss(X, Y, w, b):
    return np.average((predict(X, w, b) - Y) ** 2)
```

Look at this function's arguments. X and Y stay the same every time loss() is called. To make the upcoming discussion easier, let's also temporarily fix b at 0. So now the only variable is w.

How does the loss change as w changes? I put together a program that plots loss() for w ranging from -1 to 4, and also draws a green cross on its minimum value. (As usual, that code is amongst the book's source code).



Nice curve! Let's call it the *loss curve*. The entire idea of `train()` is to find that marked spot at the bottom of the curve—the value of `w` that gives the minimum loss. At that `w`, the model approximates the data points as well as it can.

Now imagine that the loss curve is a valley, and there is a hiker standing somewhere in this valley. The hiker wants to reach her basecamp, right where the marked spot is—but it's dark, and she can only see the terrain right around her feet. To find the basecamp, she can follow a simple algorithm: walk in the direction of the steepest downward slope. If the terrain doesn't contain holes or cliffs—and our loss function doesn't—then the hiker will eventually get close to the basecamp.

This simple idea becomes a powerful algorithm with the help of some math. Mathematicians have long known a few handy ways to calculate the slope of a curve. They call that slope the *gradient* of the curve.

The gradient of the loss curve tells us how much the loss changes when the weight changes. Imagine increasing the weight just a tiny little bit. What happens to the loss? If the gradient is a large positive number, that means that the loss is climbing steeply. If the gradient is negative, that means that the loss is decreasing. If the gradient is close to zero, that means that the loss isn't changing much at all.

At the minimum point of the curve—the one marked with the cross—the curve is level, and the gradient is zero. To the right and the left of that point, the gradient points "uphill"—it's positive if the curve is pointing upwards, and negative if the curve is pointing downwards. This means that our hiker would have to walk in the direction *opposite* to the gradient to approach the minimum.

The size of the hiker's steps should also be proportional to the gradient. If the gradient is a big number (either positive or negative), that means that the curve is steep, and the basecamp is far away. So the hiker can take big steps with confidence. As the hiker approaches the basecamp, the gradient becomes smaller, and so do her steps.

The algorithm I just described is called *gradient descent*, or GD for short. Let's brush up on our math skills and implement it.

A Sprinkle of Math

Here is how to crack gradient descent with math. First, let's rewrite the `loss()` function in good old-fashioned mathematical notation. `L` stands for the loss, and `m` stands for the number of examples:

$$L = \frac{1}{m}\sum((wx + b) - y)^2$$

If you're not familiar with this notation, then just know that the $\sum$ symbol stands for "sum". So the formula above adds the squared errors of all the examples together, and divides the result by the number of data points, to find the mean squared error.

Remember that the `xs` and `ys` are constants—they are the values of the input variable and the labels, respectively. `m` is also a constant because the number of examples never changes, and so is `b`, because we temporarily fixed it at zero. The only value that varies in the formula above is `w`.

Now we have to calculate the gradient of `L` as `w` varies. In mathspeak, that's also called "the derivative of `L` with respect to `w`," and it's written $\frac{\partial L}{\partial w}$.

Math Deep Dive: Calculus

By talking about derivatives (and soon, partial derivatives), we're touching on one of the most important branches of mathematics: *calculus*. If you want to get deeper on calculus, you'll find that Khan Academy has a long series of lessons on it.[a] As usual, it's way more material than you need here.

a. www.khanacademy.org/math/differential-calculus

If you remember calculus from school, then you might want to calculate this derivative on your own. Otherwise, don't worry about it. Somebody already did that calculation for us:

$$\frac{\partial L}{\partial w} = \frac{1}{m}\sum 2x((wx + b) - y)$$

The derivative of the loss is somewhat similar to the loss itself, except that the power of 2 is gone, and each element of the sum has been multiplied by `2x`. We can plug any value of `w` into the formula above, and we get back the value of the gradient at that point. Here's the formula above converted to code:

03_gradient/gradient_descent_without_bias.py

```
def gradient(X, Y, w):
    return np.average(2 * X * (predict(X, w, 0) - Y))
```

We fixed `b` at 0 as planned. Otherwise, `gradient()` is pretty similar to `loss()`.

Now that we have a function to calculate the gradient, we can rewrite `train()` to do gradient descent.

Downhill Riding

Here is train(), updated for gradient descent:

03_gradient/gradient_descent_without_bias.py

```
def train(X, Y, iterations, lr):
    w = 0
    for i in range(iterations):
        print("Iteration %4d => Loss: %.10f" % (i, loss(X, Y, w, 0)))
        w -= gradient(X, Y, w) * lr
    return w
```

You might be surprised by how terse train() became. With GD, we don't need any if. We just initialize w, and then step repeatedly in the opposite direction of the gradient. (Remember, the gradient is pointing uphill, and we want to go downhill). The lr hyperparameter is still there, but now it tells how large each step should be in proportion to the gradient.

One problem with gradient descent is when to stop it. The old version of train() returned after a maximum number of iterations, or when it failed to decrease the loss further—whichever came first. With GD, the loss could in theory decrease forever, inching towards the minimum in smaller and smaller steps as the gradient approaches zero. So, when should we stop making those ever-tinier steps?

We could decide to stop when the gradient becomes small enough, or maybe when the difference between the current and the previous loss becomes small enough. Both approaches are valid. We can also chicken out of this decision, and go for the quick and dirty solution: the caller tells train() how many iterations to run. More iterations lead to a lower loss, but since the loss decreases progressively more slowly, at a certain point we can just decide that the additional precision isn't worth the wait.

In a future chapter, you will learn how to pick good values for hyperparameters such as iterations and lr. For now, I just tried a bunch of different values and ended up with these ones, which seem to result in a low enough, precise enough loss:

```
if __name__ == "__main__":
    X, Y = np.loadtxt("pizza.txt", skiprows=1, unpack=True)
    w = train(X, Y, iterations=100, lr=0.001)
    print("\nw=%.10f" % w)
```

Here is what we get by running this code:

```
Iteration    0 => Loss: 812.8666666667
Iteration    1 => Loss: 304.3630879787
Iteration    2 => Loss: 143.5265791020
```

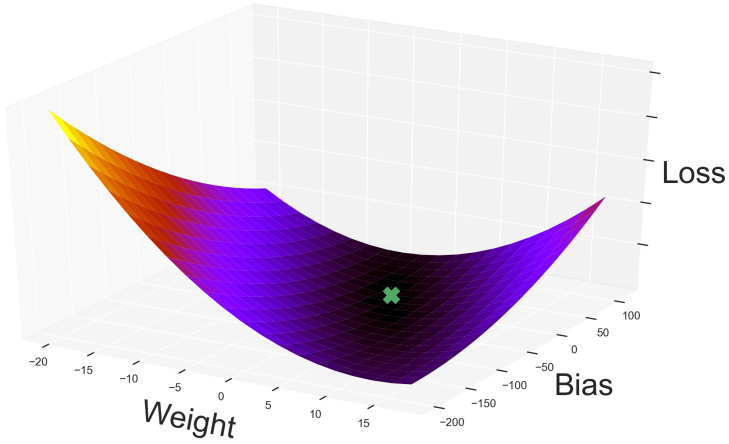
```
…
Iteration   98 => Loss: 69.1209628275
Iteration   99 => Loss: 69.1209628275

w=1.8436928702
```

With 100 iterations, the precision is already high enough that you can't see the difference between the last two losses. The result is a lower loss than we got with the old, hacky code in Let's Do This!, on page 25, even though we've been running for fewer iterations. Also, each of those iteration is calling `loss()` only once rather than four times. We got better results, at a lower computational cost. It seems like we're headed in the right direction.

But wait—remember that we're only using the `w` parameter. Let's put `b` back in the game and see where that leads us.

Walking the Land of Loss

Let's see what happens when we change `b` back to a variable. We've seen the loss curve with one variable `w`. If we put `b` back in, the loss is not a two-dimensional curve anymore. Now it's a surface:



The two horizontal axes are the values of `w` and `b`, and the vertical axis is the value of the loss. Now our hiker doesn't live in flatland anymore—she's free to walk around in three dimensions. The basecamp, once again marked with a cross, is the lowest point in the entire surface. Let's see how to reach it.

Here is the loss function in math notation again:

$$L = \frac{1}{m}\sum((wx + b) - y)^2$$

So far, we treated all the values in this function as constants, except for `w`. This time, we have two variables: `w` and `b`. Lucky for us, there's an easy way to calculate the gradient of a function of multiple variables. It works like this.

First, for each variable, pretend that it's the *only* variable in the function. Just imagine that everything else is constant, and calculate the derivative with respect to that sole variable. We already did that in A Sprinkle of Math, on page 36: we fixed `b` and calculated $\frac{\partial L}{\partial w}$, the derivative of `L` with respect to `w`. Now we need to do the same with the other variable.

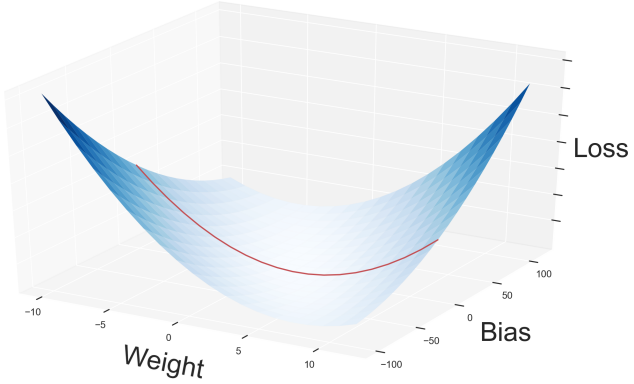
Let's pretend that `w` is a constant, and take the derivative of `L` with respect to `b`. Once again, those of you who studied calculus can come up with this derivative on their own. For the rest of us, here it is:

$$\frac{\partial L}{\partial b} = \frac{1}{m} \sum 2((wx + b) - y)$$

This idea of taking a function of multiple variables and calculating its derivative with respect to a single variable is called a *partial derivative*. You don't need to know much about partial derivatives to finish this book, but here is an intuitive explanation for the curious reader.

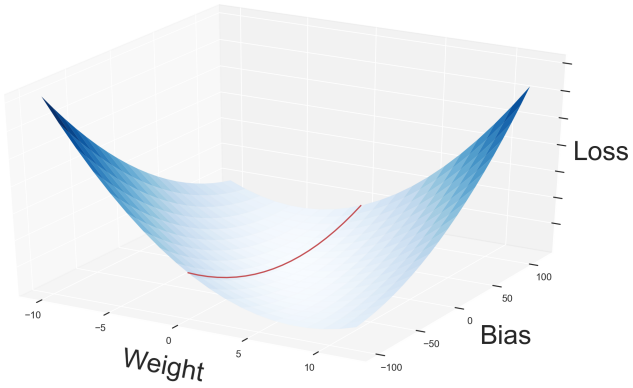
Partial Derivatives

Let's see what partial derivatives are, and how they can help us. Taking a partial derivative is like slicing a function with a katana, and then calculating the gradient of the slice. (It doesn't have to be a katana, but using a katana makes the process look cooler.) For example, when we fixed `b` at 0, we sliced the loss like this:

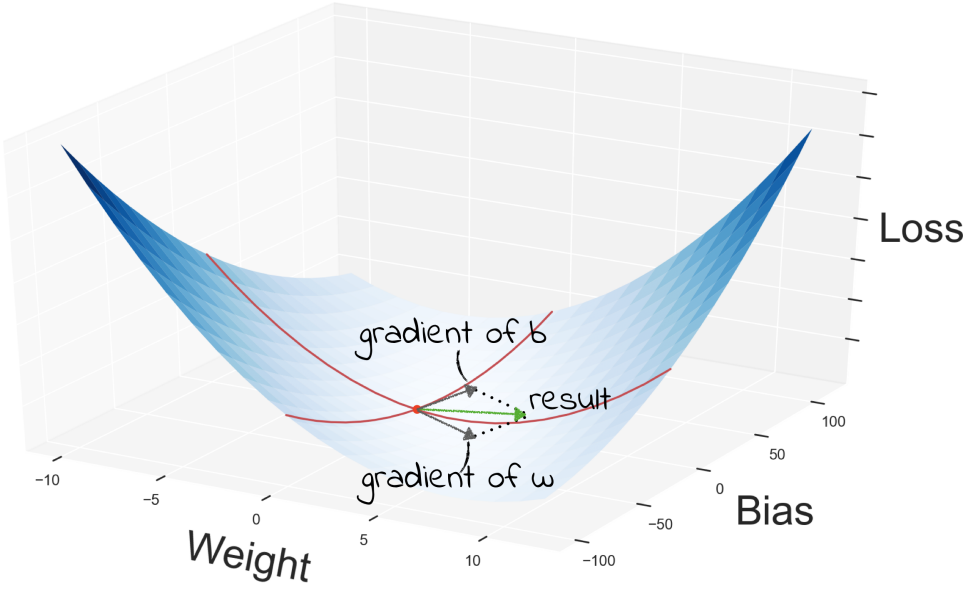


The slice shown above is the same loss curve that we plotted in Gradient Descent, on page 35. It looks squashed because I used different ranges and scales for the axes, but it's exactly the same function. For each value of `b`,

you have one such curve, which has `w` as its only variable. And of course, for each value of `w` you have a similar curve with variable `b`. Here is the one with `w = 0`:



Now that we have those 1-dimensional slices, we can calculate the gradients on them, just like we did for the original curve. And here's the good news: if we combine the gradient of the slices, then we get the gradient of the surface:



Thanks to partial derivatives, we just split a 2-variables problem into two 1-variable problems. That means that we don't need a new algorithm to do GD on a surface. Instead, we just slice the surface using partial derivatives, and then do GD on each slice.

Let's recap the entire process. Our hiker is standing at a specific spot—that is, a specific value of `w` and `b`. She's armed with the formulas that we found

earlier to compute the partial derivatives of `L` with respect to either `w` or `b`. She plugs the current values of `w` and `b` into those formulas, and she gets one gradient for each variable. She applies GD to both gradients, and there you have it! She's descending the gradient of the surface.

Enough math for this chapter. Let's put this algorithm in code.

Putting Gradient Descent to the Test

It's time to put together the two-variables version of our gradient descent code. Here it is, with the changes marked by little arrows in the left margin:

03_gradient/gradient_descent_final.py

```
import numpy as np

def predict(X, w, b):
    return X * w + b

def loss(X, Y, w, b):
    return np.average((predict(X, w, b) - Y) ** 2)

➤ def gradient(X, Y, w, b):
➤     w_gradient = np.average(2 * X * (predict(X, w, b) - Y))
➤     b_gradient = np.average(2 * (predict(X, w, b) - Y))
➤     return (w_gradient, b_gradient)

def train(X, Y, iterations, lr):
➤     w = b = 0
      for i in range(iterations):
➤         print("Iteration %4d => Loss: %.10f" % (i, loss(X, Y, w, b)))
➤         w_gradient, b_gradient = gradient(X, Y, w, b)
➤         w -= w_gradient * lr
➤         b -= b_gradient * lr
➤     return w, b

if __name__ == "__main__":
      X, Y = np.loadtxt("pizza.txt", skiprows=1, unpack=True)
➤     w, b = train(X, Y, iterations=20000, lr=0.001)
      print("\nw=%.10f, b=%.10f" % (w, b))
      print("Prediction: x=%d => y=%.2f" % (20, predict(20, w, b)))
```

The `gradient()` function now returns the partial derivatives of the loss with respect to both `w` and `b`. Those gradients are used by `train()` to update both `w` and `b`, at the same time. I also bumped up the number of iterations, because the program takes longer to get close to the minimum now that it has two variables to tweak.

Let's make an apples-to-apples comparison between this new version of the program and the one from the previous chapter. First, let's run the earlier

version with plenty of iterations and a pretty low lr of 0.0001, to get four decimal digits of precision:

```
…
Iteration 157777 -> Loss: 22.842737

w=1.081, b=13.171
Prediction: x=20 => y=34.80
```

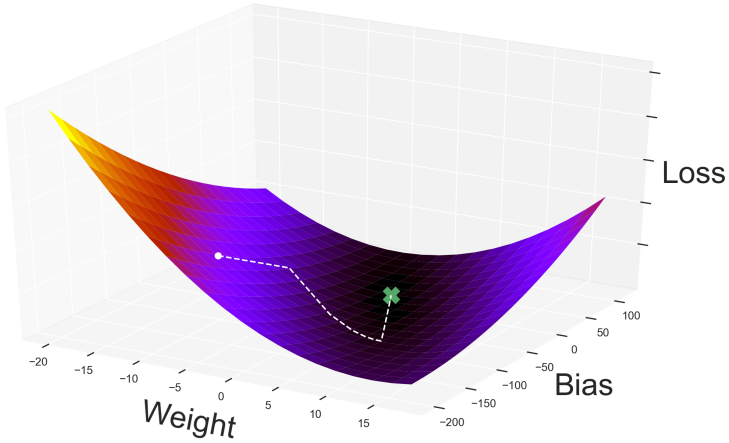
Our new GD-based implementation runs circles around that result. With just 20,000 iterations, we get:

```
…
Iteration 19999 => Loss: 22.8427367616

w=1.0811301700, b=13.1722676564
Prediction: x=20 => y=34.79
```

Higher precision in almost one tenth of the iterations. Good stuff! This faster, more precise learner might be wasted for our pizza forecasting problem—after all, nobody buys one hundredth of a pizza. However, that additional speed will prove essential to tackle more advanced problems.

The Smooth Way Home

You should have an intuitive understanding of gradient descent by now, but nothing beats watching it in motion. Here is what we get by plotting the algorithm's path, from an arbitrary starting point to the minimum loss:



The hiker didn't take the optimal route to the basecamp, because she couldn't know that route in advance. Instead, she let the slope of the loss function guide her every step. After two abrupt changes of direction, she finally reached the bottom of the valley, and proceeded along a gently sloping trail down to the basecamp.

When Gradient Descent Fails

GD doesn't give you many guarantees. By using it, we could follow a longer route than the shortest possible one. We could step *past* the basecamp, and then have to backtrack. We could even step further away from the basecamp, as the crow flies. But GD does guarantee that we will eventually reach our target… as long as the terrain meets certain conditions.

With some imagination, you can probably think of a surface that makes GD fail. What if our hiker gets stuck in a hole—a spot higher than the basecamp, but lower than the terrain around it? (That's called a *local minimum*, as opposed to the *global minimum* that we want to reach.) What if the loss function has a sudden cliff, so that our hiker ends up making her best Wile E. Coyote impression?

In math-speak, a good loss function should be *convex*, meaning that it shouldn't have bumps that result in local minima; *continuous*, meaning that it doesn't have vertical cliffs or interruptions; and *differentiable*, meaning that it should be smooth, without cusps and other weird spots where you cannot even calculate a gradient. Our current loss function ticks all those boxes, so it's ideal for GD. Later on, we will apply GD to other functions, and we will vet those functions for such prerequisites.

GD is also the main reason why we implemented our loss with the mean squared error formula. We could have used the mean _absolute value_ of the error—but the mean absolute value doesn't work well with GD, because it has a cusp around the value 0. As a bonus, squaring the error makes large errors even larger, creating a really steep surface as you get further away from the minimum. In turn, that steepness means that GD blazes towards the minimum at high speed. Because of both its smoothness and its steepness, the mean squared error is a great fit for GD.

What You Just Learned

In this chapter we investigated *gradient descent*, the most widely used algorithm to miminize loss. It can be described with one sentence: take a step in the opposite direction as the *gradient* of the loss, and keep doing that until the gradient becomes small. To find the gradient, we took the *partial derivatives* of the loss with respect to `w` and `b`.

The biggest limitation of GD is that it can get stuck in a *local minimum*, failing to reach the *global minimum*. To avoid that problem, we'll try to use loss functions that have only one minimum.

GD is not the be all and end all of the algorithms that minimize loss. Researchers are exploring alternative algorithms that do better in some circumstances. There are also variations of plain vanilla GD, some of which we'll meet later in this book. Nonetheless, GD is still the foundation of modern machine learning, and it's likely to keep that spot for a while.

And now, prepare for a challenge. At the beginning of this chapter, I told you that GD allows our code to scale to more interesting problems that involve many parameters. One such problem is coming our way in the next chapter.

Hands On: Basecamp Overshooting

If you closed the previous chapter by tweaking the learning rate, now try the same with the `iterations` argument as well. How do these two arguments interact? You might notice that a large `lr` sometimes causes the loss to *increase* instead of decreasing. Can you imagine why?

If you can't picture the answer in your mind, try drawing the loss function on paper. What happens with a very large learning rate?

CHAPTER 4

Hyperspace!

In the previous two chapters, we solved a simple program: we predicted a restaurant's pizza sales from its reservations. Most interesting real-world problems, however, have more than one input variable. Even something as simple as pizza sales isn't likely to depend on reservations *alone*. For example, if there are many tourists in town, then the restaurant will probably sell more pizzas, even if it got as many reservations as yesterday.

If pizza sales have many variables, then imagine how many variables we'll have to consider once we get into complex domains like recognizing pictures. A learning program that only supports one variable is never going to solve those hairy problems. If we ever want to tackle them, then we'd better upgrade our program to support multiple variables.

We can learn from multiple input variables with a souped-up version of linear regression—one called *multiple linear regression*. In this chapter, we will extend our program to support multiple linear regression. We'll also add a few tricks to our bag, including a couple of useful matrix operations and several NumPy functions. Let's dive right in!

A Tough Chapter

Public service announcement: For most readers, this may be the hardest chapter in the entire book. There are two reasons for that difficulty. First, the next few pages are relatively heavy on math, and they could be intimidating unless you're familiar with linear algebra. Second, the code in this chapter is minimal, but deep. At one point, we'll go through almost two pages to explain a single line of code.

Don't let those challenges put you off. The concepts and techniques in this chapter are pivotal to ML, and very much worth learning. (If you find this little pep talk unconvincing, then consider this: after this chapter, the rest of the book will look easy!)

Adding More Dimensions

In the previous chapter, we coded a gradient descent-based version of our learning program. This souped-up program can potentially scale to complex models with more than one variable.

In a moment of weakness, we mentioned that opportunity to our friend Roberto. That was a mistake. Now Roberto is all pumped up about forecasting pizza sales from a bunch of different input variables besides reservations, such as the weather, or the number of tourists in town.

This is going to be more work for us—and yet, we can't blame Roberto for wanting to add variables to the model. After all, the more variables we consider, the more likely it is that we'll get accurate predictions of pizza sales.

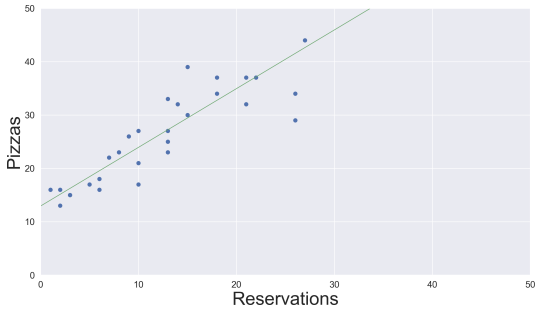
Let's start with a souped-up version of the old pizza.txt file. Here are the first few lines of this new dataset, more_pizza.txt:

04_hyperspace/more_pizza.txt

```
Reservations  Temperature   Tourists  Pizzas
13            26            9         44
2             14            6         23
14            20            3         28
```

The first three columns in the file contain input variables, and the last one contains labels. We already know the first and last column: reservations and pizzas sold, respectively. The second column is new: Roberto suspects that more people come drop into his pizzeria on warmer days, so he kept track of the temperature in degrees Celsius. (For reference, 2 °C is almost freezing; 26 °C is about 78 Fahrenheit). The third column is the density of tourists in town, downloaded from the local tourist office's website. It ranges from 1 ("not a soul in town") to 10 ("tourist invasion").

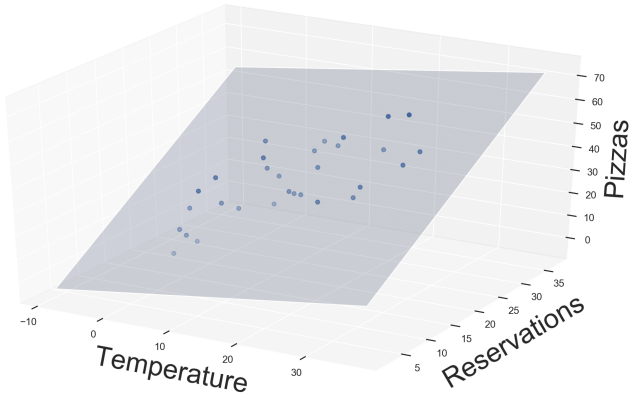
Let's see what happens to linear regression once we have multiple input variables. You know that linear regression is about fitting a line to the examples:



As a reminder, here is the formula of that line:

```
ŷ = x * w + b
```

If we add a second input variable (say, "temperature"), then the examples aren't laying on a plane anymore—they're points in three-dimensional space. To fit them, we can use the equivalent of a line, with one more dimension: a plane.



Like we did for the line, we can use the equation of the plane to calculate ŷ. Here is what it looks like:

```
ŷ = x1 * w1 + x2 * w2 + b
```

The equation of a plane is similar to the equation of a line—but it has two input variables, `x1` and `x2`, and two weights, `w1` and `w2`.

If you're not convinced that we need a separate weight for each input variable, try thinking in more concrete terms. In our example, `x1` is the number of reservations and `x2` is the temperature. It makes sense that the reservations and the temperature have different impacts on the number of pizzas, so they must have different weights.

In the equation of a line, the bias `b` shifts the line away from the origin. The same goes for a plane: if we didn't have `b`, then the plane would be constrained to pass by the origin of the axes. If you want to prove that, just set all the input variables to 0. Without a bias, $\hat{y}$ would also be 0. Thanks to the bias, the plane is free to shift vertically and find the position where it approximates the points as well as it can.

We started by fitting a one-dimensional line to bi-dimensional examples. Then we added an input variable, and we moved on to fit a bi-dimensional plane to three-dimensional examples. Add more input variables, and this process continues: with `n` input variables, we must fit an `(n - 1)`-dimensional shape to `n`-dimensional examples.

With the exception of H. P. Lovecraft's characters, humans cannot perceive more than three dimensions. However, math has no problem dealing with those sanity-bending multi-dimensional spaces—it just calls them *hyper-spaces*, and describes them with the same equations as bi-dimensional and three-dimensional spaces. However many dimensions we have, we can just add input variables and weights to the formula of the line and the plane:

```
ŷ = x1 * w1 + x2 * w2 + x3 * w3 + … + b
```

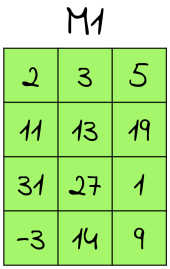
The formula above is called the *weighted sum* of the inputs. The equation of a line is just a special case—the weighted sum of a single input. So, here's a simple plan to upgrade our learning program from one to many input variables: we'll replace the equation of a line with the more generic formula of the weighted sum.

To implement that simple plan, however, we have to know about a couple more mathematical operations. Let's take a short detour to learn them.

Matrix Math

To upgrade our code, we'll need to know about two important operations on matrices. I don't want to introduce those operations at the last moment, while we're busy coding. Let's take two or three pages to get them out of the way now.

First, let me make it clear what I mean by "matrix." As a programmer, you know that a *matrix* is an array with more than one dimension. In math, a matrix is even simpler. It's like a bi-dimensional table:



`M1` is a `(4, 3)` matrix, meaning that it has 4 rows and 3 columns.

The two operations that we'll cover are *matrix multiplication* and *matrix transpose*. Both are ubiquitous in ML, and each deserves its own section.

Math Deep Dive: Linear Algebra

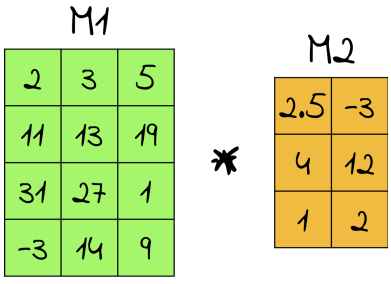
Matrix operations such as multiplication and transpose belong to a branch of math called *linear algebra*. As usual, you can take a much deeper look at linear algebra on Khan Academy.[a]

a. www.khanacademy.org/math/linear-algebra

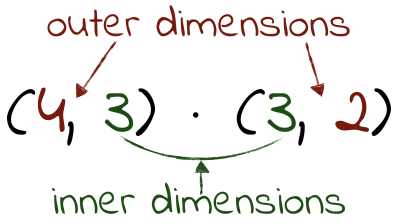
Multiplying Matrices

Did you ever wonder why ML is usually done on those big racks of GPUs? That's because ML systems spend most of their time doing one operation, that happens to be particularly fast on a GPU: they multiply big matrices.

To introduce *matrix multiplication*, let me tell you its "golden rule" first: we can multiply two matrices if (and only if) the second matrix has as many rows as the first matrix has columns, like this:



`M1` is `(4, 3)` and `M2` is `(3, 2)`. Can we multiply them? To answer this question, write down the operation like this: `(4, 3) · (3, 2)`. Of the four dimensions in the operation, ignore the outer ones, and look at the inner ones:



If the inner dimensions are equal, then we can multiply the matrices. In our case, they are both 3, so yes, we can multiply `M1` by `M2`.

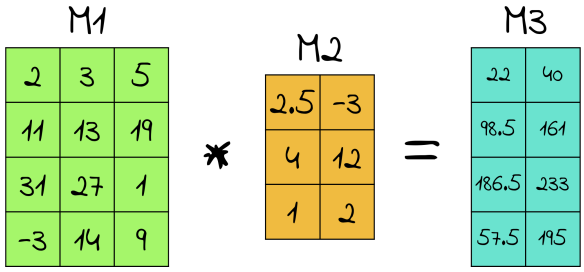
To build up to matrix multiplication, let's first define an intermediate operation called *dot product*. Pick any row from `M1` and any column from `M2`. The row and column have the same number of elements, because of the golden rule. The dot product of the row and the column is a single number calculated like this:

```
row · column = first_element_of_row * first_element_of_column +
               second_element_of_row * second_element_of_column +
               ... +
               last_element_of_row * last_element_of_column
```

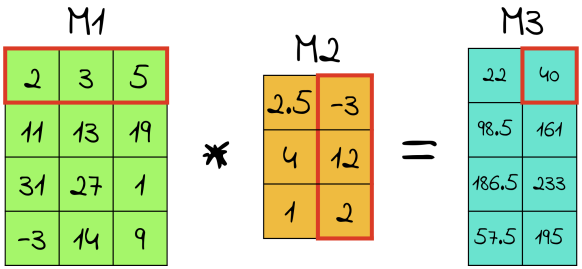
Now that we have the dot product, we can finally define matrix multiplication. If we multiply `M1` by `M2`, then we get a matrix `M3` where each element `(i, j)` in `M3` is the dot product of row `i` from `M1` and column `j` from `M2`:

```
M3[i][j] = i-th_row_of_M1 · j-th_column_of_M2
```

Here comes a concrete example. I calculated the multiplication of `M1` by `M2`:



Let's double-check one of `M3`'s elements—say, `M3[0][1]`, which is 40. The rule says that `M3[0][1]` should be equal to row 0 of `M1` by column 1 of `M2`:



The dot product of that row and column is `2 * -3 + 3 * 12 + 5 * 2 = 40`, as we expected. Repeat the process for each element of `M3`, and there you have it—matrix multiplication.

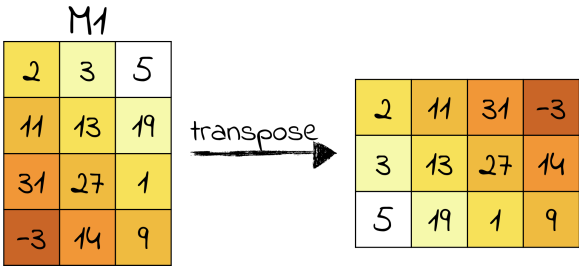
Note that in matrix multiplication, different than regular multiplication, order matters. If we swap `M1` and `M2`, then we generally get a different result, and in most cases we cannot complete the multiplication at all. For example, we couldn't multiply `M2` by `M1`, because the inner dimensions in $(3, 2) \cdot (4, 3)$ differ.

We'll have plenty of matrix multiplications in this book, but we'll use NumPy to calculate the results rather than do it by hand. However, there are two things about matrix multiplication that you should remember by heart. First, the golden rule: you can only multiply two matrices if the inner dimensions in the multiplication are equal. And second, the shape of the result: the result of matrix multiplication has as many rows as the first matrix, and as many columns as the second matrix. In our example, we multiplied $(4, 3) \cdot (3, 2)$, so the result is a $(4, 2)$ matrix.

And that's how matrix multiplication works. Let's move on to matrix transpose.

Transposing Matrices

Compared to matrix multiplication, *matrix transpose* is easy. When you transpose a matrix, you flip it around the diagonal that goes from top left to bottom right. This diagram uses graduated cell colors to show what a transposed matrix looks like:



Transposing a matrix means that row data becomes column data, and the other way round. As a result, the matrix's dimensions are swapped. The matrix above is (4, 3). When we transpose, it becomes (3, 4).

Now you know about matrix multiplication and transpose. Both operations will come useful in the next few pages. Let's go back to the code.

Upgrading the Learner

After this mathematical detour, we can come back to the work at hand. We want to upgrade our learning program to deal with multiple input variables. Let's make a plan of action so that we don't get lost in the process:

- First, we'll load and prepare the multi-dimensional data, so that we can feed it to the learning algorithm.
- After preparing the data, we'll upgrade all the functions in our code to use the new model: we'll switch from a line to a more generic weighted sum, as I described in Adding More Dimensions, on page 48.

Let's step through this plan. Feel free to type the commands in a Python interpreter and make your own experiments. If you don't have an interpreter handy, then fear not: I will show you the output of all important commands (sometimes slightly edited, to spare space).

Preparing Data

I wish I could tell you that machine learning is all about building amazing AIs and looking cool. The reality is that a large part of the job is preparing data for the learning algorithm. So far, we just called NumPy's loadtxt(), and that was it. Now we must massage the data a bit afterwards.

Once again, here are the first lines of more_pizza.txt:

04_hyperspace/more_pizza.txt

```
Reservations  Temperature   Tourists  Pizzas
13            26            9         44
2             14            6         23
14            20            3         28
```

We have four columns in the file, so loadtxt() will return four NumPy arrays instead of two. The first three arrays contain the input variables (reservations, temperature and tourist density). The last array, as usual, contains the labels:

```
import numpy as np
x1, x2, x3, y = np.loadtxt("more_pizza.txt", skiprows=1, unpack=True)
```

Arrays are NumPy's killer feature. They're very flexible objects that can contain anything from a *scalar* (a single number), to a multi-dimensional matrix. To see which dimensions an array has, we can use its shape() operation. We'll use it a lot.

```
x1.shape  # => (30, )
```

All four columns have 30 elements, one for each example in more_pizza.txt. That dangling comma is NumPy's way of saying that these arrays have just one dimension. In other words, they're what you probably think about when you hear the word "array." Array indexing also works as you expect:

```
x1[2]  # => 14.0
```

Indexes in NumPy are zero-based, so this is the third element in the first column (x1), that contains reservations.

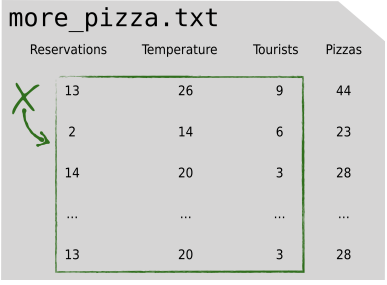
To make the data easier to deal with, let's join the three arrays of input variables into a matrix:

```
X = np.column_stack((x1, x2, x3))
X.shape  # => (30, 3)
```

Here are the first two rows of X (the notation [:2] means the same thing as [0:2]: "from index zero to 2, excluded"):

```
X[:2]  # => array([[13., 26., 9.], [2., 14., 6.]])
```

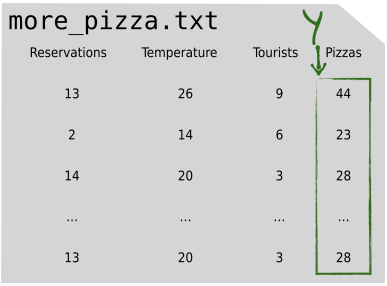
X contains the rows and columns from more_pizza.txt, minus the labels. Each row is an example, and each column is an input variable:



Now that we took care of X, let's look at y, that still has that one-dimensional (30,) shape. Here is one trick that saved my bottom multiple times: avoid mixing NumPy matrices and one-dimensional arrays. Code that involves both can have surprising behavior. For this reason, as soon as you have a one-dimensional array, it's a good idea to reshape it into a matrix with the reshape() function:

```
Y = y.reshape(-1, 1)
```

reshape() takes the dimensions of the new array. If one of the dimensions is -1, then NumPy will set it to whatever makes the other dimensions fit. So the line above means: "reshape Y so that it's a matrix with 1 column, and as many rows as you need to fit the current elements". The result is a (30, 1) matrix:




```
Y.shape  # => (30, 1)
```

With that, we're done preparing data. Let's move to update the functions of our learning system, starting with the predict() function.

Upgrading Prediction

Now that we have multiple input variables, we need to change the prediction formula—from the simple equation of a line, to a weighted sum:

$$\hat{y} = x1 * w1 + x2 * w2 + x3 * w3 + \ldots$$

To simplify the formula, I temporarily removed the bias b. It will be back soon.

Now we can translate the weighted sum above to a multi-dimensional version of predict(). As a reminder, here is the old mono-dimensional predict() (without the bias):

```
def predict(X, w):
    return X * w
```

The new predict() should still take X and w—but those variables have more dimensions now. X used to be a vector of m elements, where m is the number of examples. Now X is a matrix of (m, n), where n is the number of input variables. In the specific case that we're tracking now, we have 30 examples and 3 input variables, so X is a (30, 3) matrix.

What about w? Just as we need one x per input variable, we also need one w per input variable. Different from the xs, though, the ws will be the same for each example. So we could make the weights a matrix of (n, 1), or a matrix of (1, n). For reasons that will become clear in a minute, it's better to make it (n, 1): one row per input variable, and a single column.

Let's initialize this matrix of (n, 1). Do you remember that we used to initialize w at zero? Now w is a matrix, so we must initialize all its elements to zeros. NumPy has a zeroes() function for that:

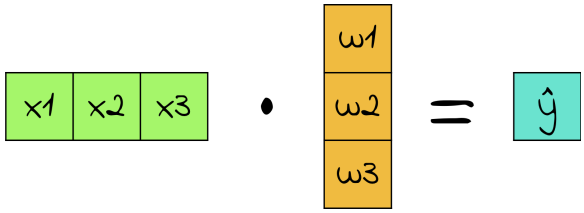
```
w = np.zeros((X.shape[1], 1))
w.shape  # => (3, 1)
```

X.shape[1] is the number of columns in X, which is the number of input variables—in our case, 3.

And this is where matrix multiplication finally becomes useful. Look back at the weighted sum:

```
ŷ = x1 * w1 + x2 * w2 + x3 * w3
```

That's exactly the same operation as multiplying one row of X by w:



What if we have multiple examples? Then X has many rows—say, (30, 3). So when we multiply it by w, which is (3, 1), we get a (30, 1) matrix. That's a matrix with one row per example, and a single column that contains the prediction for that example. We can get the predictions for all examples in one fell swoop!

So it turned out that rewriting prediction for multiple input variables is as simple as multiplying X and w. We can do that with NumPy's matmul() function:

```
y_hat = np.matmul(X, w)
y_hat.shape  # => (30, 1)
```

This new way to calculate prediction took a while to explain, but the final result is really short and simple: the new predict() will be just the matrix multiplication of X and w. Sweet!

Upgrading the Loss

Moving on to the loss() function—remember that we used the mean squared error to calculate the loss, like this:

```
def loss(X, Y, w):
    return np.average((predict(X, w) - Y) ** 2)
```

Once again, I'm using the version of the loss without a bias, so that we have one less thing to worry about. We'll reintroduce the bias soon. For now, let's

see how we should change the loss() function above to accomodate multiple dimensions.

I remember how frustrated I got with matrix operations when I wrote my first ML programs. Matrix dimensions, in particular, never seemed to add up. With time, I learned that matrix dimensions could actually be my friends: if I tracked them carefully, they could help me piece my code together. Let's do the same here: we can use matrix dimensions to guide us through the code.

Both y_hat and Y are (m, 1) matrices, meaning that they have one row for example, and one column. In our case, we have 30 examples, so they're (30, 1). If we subtract Y from y_hat, NumPy checks that the two matrices have the same size, and subtracts each element of Y from each element of y_hat. The result is still (30, 1).

Then we square the result, and NumPy dutifully applies the "square" operation to each element of the resulting matrix. This is a feature of NumPy called "broadcasting", which we already used in the previous chapters: when we apply an arithmetic operation to a NumPy array, the operation is "broadcast" over each element of the array. The result of this operation is, once again, (30, 1).

Finally, we call average(), that averages all the elements in the matrix, returning a single scalar number:

```
a_number = loss(X, Y, w)
a_number.shape  # => ()
```

The empty parentheses are NumPy's way of saying: "this is a scalar, so it has no dimensions."

Here's the bottom line: we don't need to change the way we calculate the loss at all. Our mean squared error code works with multiple input variables just as well as it did with one variable.

We just have one last thing to change.

Upgrading the Gradient

Let me cut straight to the chase. Here's how we calculate the gradient with multiple input variables:

```
def gradient(X, Y, w):
    return 2 * np.matmul(X.T, (predict(X, w) - Y)) / X.shape[0]
```

X.T means "X transposed"—the operation that we talked about in Transposing Matrices, on page 53.

The code above is the same as the old gradient, only with multiple input variables. I'm not going to prove that fact mathematically, because math proofs are outside the scope of this book. If you're an expert at linear algebra, you can do it yourself—it's not too hard, and it could even be fun. Otherwise, you can just take this code for granted.

Double-Checking the Gradient

Most bugs in matrix-related code can be spotted early by checking all the matrix dimensions. When we upgraded loss(), we stepped through each check together. In the case of the code in Upgrading the Gradient, on page 58, I'll leave those checks to you alone.

You'll probably want to arm yourself with pen and paper, or a Python interpreter. Here are a few hints:

- To subtract two matrices, they must have the same shape.
- Remember the stuff we talked about in Multiplying Matrices, on page 51—in particular, the "golden rule" of matrix multiplication, and the shape of the result.
- In the train() function, we subtract the gradient from the weight—so they must have the same shape. While upgrading predict(), we found that the weights are (3, 1). Check that the matrix returned by gradient() is also (3, 1).

Now that we have each individual function, we can put together a program that does multiple linear regression.

Putting It All Together

Let's check that we have everything in order:

- we wrote the code that prepares the data;
- we upgraded predict();
- we came to the conclusion that there is no need to upgrade loss();
- we upgraded gradient().

Check… check… check… and check. We can finally apply all those changes to our learning program:

04_hyperspace/multiple_regression_without_bias.py

```
import numpy as np

def predict(X, w):
    return np.matmul(X, w)

def loss(X, Y, w):
    return np.average((predict(X, w) - Y) ** 2)
```

```
def gradient(X, Y, w):
    return 2 * np.matmul(X.T, (predict(X, w) - Y)) / X.shape[0]

def train(X, Y, iterations, lr):
    w = np.zeros((X.shape[1], 1))
    for i in range(iterations):
        print("Iteration %4d => Loss: %.20f" % (i, loss(X, Y, w)))
        w -= gradient(X, Y, w) * lr
    return w

if __name__ == "__main__":
    x1, x2, x3, y = np.loadtxt("more_pizza.txt", skiprows=1, unpack=True)
    X = np.column_stack((x1, x2, x3))
    Y = y.reshape(-1, 1)
    w = train(X, Y, iterations=500000, lr=0.001)
```

The main code changed the most because it needs to prepare data. Apart from that, we changed just three lines. The specific problem that we're solving has three input variables in our example, but all the functions are generic enough to work with any number of input variables.

If we run the program, here's what we get:

```
Iteration    0 => Loss: 1333.56666666666660603369
Iteration    1 => Loss: 151.14311361881479456315
Iteration    2 => Loss: 64.99460808656145616169
…
Iteration 499999 -> Loss: 6.89576133146784187034
```

The loss decreases at each iteration, and that's a hint that the program is indeed learning. However, we know by now that we shouldn't expect good predictions without a bias parameter. Fortunately, putting the bias back in is easier than it looks.

Bye Bye, Bias

So far, we implemented this prediction formula:

```
ŷ = x1 * w1 + x2 * w2 + x3 * w3
```

Now we want to add the bias back to the system, like this:

```
ŷ = x1 * w1 + x2 * w2 + x3 * w3 + b
```

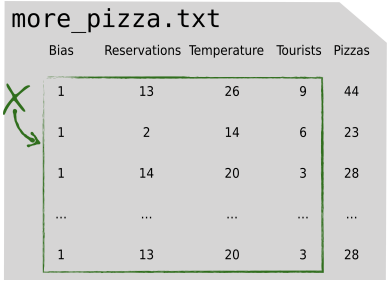
We could rush to the code and add the bias everywhere, like we had in the previous chapter—but hold on a minute. I can teach you a trick to roll the bias into the code more smoothly.

Give another look at the previous formula. What's the difference between the bias and the weights? The only difference is this: the weights are multiplied by some input variable `x`, and the bias is not. Now imagine that there is one more input variable in the system (let's call it `x0`) that always has a value of 1. We can rewrite the formula like this:

```
ŷ = x1 * w1 + x2 * w2 + x3 * w3 + x0 * b
```

Now there's no difference at all between bias and weights. The bias is just the weight of an input variable that happens to have the constant value 1. So, here's the trick I was talking about: we can add a dummy input variable with the constant value 1, and we won't need an explicit bias anymore.

We could add this constant input variable by adding a column of 1s to the more_pizza.txt file, like this:



However, it's generally a good idea to avoid messing with the original data. Instead, let's insert a column of ones into `X` *after* we load the data. The position of this *bias column* doesn't really matter, but it's a common convention to insert it as the first column, like this:

04_hyperspace/multiple_regression_final.py

```
if __name__ == "__main__":
    x1, x2, x3, y = np.loadtxt("more_pizza.txt", skiprows=1, unpack=True)
    X = np.column_stack((np.ones(x1.size), x1, x2, x3))
    Y = y.reshape(-1, 1)
    w = train(X, Y, iterations=500000, lr=0.001)

    print("\nWeights: %s" % w.T)
    print("\nA few predictions:")
    for i in range(5):
        print("X[%d] -> %.4f (label: %d)" % (i, predict(X[i], w), Y[i]))
```

I took this chance to add a few printouts to the program. First, they print the weights matrix (transposed, to fit it on a single line). Then they print the predicted values and labels for the first five examples, so that we can compare them.

With that, our multiple linear regression program is complete. And now…

A Final Test Drive

If we run the program, we get this output:

```
Iteration    0 => Loss: 1333.5666666666660603369
Iteration    1 => Loss: 152.37148173674074769224
…
Iteration 499999 -> Loss: 6.69817788012986436996

Weights: [[ 2.41470983  1.23376707 -0.02710596  3.1246369 ]]

A few predictions:
X[0] -> 45.8707 (label: 44)
X[1] -> 23.2506 (label: 23)
X[2] -> 28.5192 (label: 28)
X[3] -> 58.2354 (label: 60)
X[4] -> 42.8002 (label: 42)
```

First, look at the loss. As we expected, it's lower than the one that we got without a bias.

The weights are interesting in their own right. The first weight is actually the bias, which we turned into a regular weight with the "column of ones" trick. The remaining weights match the three input variables—respectively reservations, temperature and tourist density. Tourist density has the biggest weight, and that means that it has the biggest correlation with pizza sales—even more than reservations. The correlation between temperature and pizzas is negligible. Such a small weight is probably just random noise, so it seems that the temperature doesn't have any effect on pizza sales at all.

Finally, the last few lines of output show predictions and labels for the first five examples. No prediction is more than a pizza or two off the mark. It seems that Roberto was right: upgrading to multiple variables boosted our ability to forecast pizzas.

Congratulations! You worked through the hardest chapter in this book. Let's wrap up what you learned.

What You Just Learned

This chapter was all about *multiple linear regression*. We extended our program to multiple input variables—using multiple weights to match the input variables. We also got rid of the explicit bias, turning it into just another weight. Our learning program is now powerful enough to tackle real-world problems, although it doesn't look any more complicated than it used to.

Along the way, you also learned about *matrix multiplication* and *matrix transpose*. These pieces of math sit right at the core of ML. Now that they're in your toolbox, they'll serve you well for years to come.

Finally, in this chapter we started delving deeper into NumPy. I personally have a love-and-hate relationship with NumPy: I love its power, but I keep getting confused by its interface. Say what you want, NumPy is a must-have ML tool, so it's important to get familiar with it. We'll keep using it throughout this book.

It's time for a plot twist: everything that we talked about in these first few chapters was just groundwork for something different and cooler. In the next chapter, we'll abandon the beaten track of linear regression, and set on the road less traveled… the road that leads to computer vision.

Hands On: Field Statistician

Now that we have a program that deals with multiple variables, you might want to try it on real-life data. In the `data/life-expectancy` directory of the book's source code, you'll find a dataset that lists life expectancy in different countries. In the same directory you'll also find a `readme.txt` with istructions.

Have fun playing with real data!

CHAPTER 5

A Discerning Machine

We started our journey in machine learning by way of linear regression. Now we're going to use that knowledge (and that code) to branch off towards our goal: a program that recognizes images.

This chapter covers the first step towards image recognition. We'll build a *binary classifier*—a program that assigns data to one of two *classes*, or groups. As an example, consider a system that diagnoses pneumonia in X-ray scans. That system would assign each scan to either a "no pneumonia" class, or a "pneumonia" class.

Where linear regression forecasts a continuous quantity that can take any decimal value, a binary classifier must forecast either 0 or 1. Just like they invented linear regression, statisticians came up with a similar technique to forecast binary values. It's called *logistic regression*, where "logistic" is just another word for "binary."

In the next few pages, we'll replace the linear regression in our program with logistic regression, turning the program into a binary classifier. Then, in the next chapter, we'll apply that binary classifier to images.

Before we dive in, be aware of a slight change in vocabulary. So far, I said that a learning system works in two phases: training and prediction. From now on, I'll call those two phases "training" and "classification" instead, to emphasize that the result of our prediction is a discrete label. "Classification" is just a specific type of prediction, so I'll use the more specific term.

Where Linear Regression Fails

Our friend Roberto asked us to help him one last time. (He *swears* it's the last time.) Roberto's pizzeria usually makes a tidy profit, but it also has bad days when it fails to break even. If only he knew in advance that a day is at

risk of not breaking even, Roberto could set up countermeasures—such as stand by the entrance and invite people inside. He suspects that the same input variables that impact the number of pizzas sold, such as temperature and tourists, also affect how well the pizzeria does overall. Maybe we can forecast bad days from those variables?

As usual, Roberto sent us a file of labeled data. Here are the first few lines:

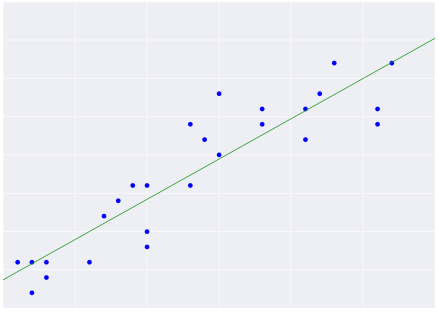
05_discerning/break_even.txt

```
Reservations  Temperature   Tourists  Break-even
13            26            9         1
2             14            6         0
14            20            3         1
23            25            9         1
13            24            8         1
1             13            2         0
```

This is the same data that we used to train our linear regression program, except for one difference: the labels are either 1, meaning that the pizzeria made a profit on that day, or 0, meaning that it failed to break even. Roberto would like a system that forecasts those binary values in the last column.

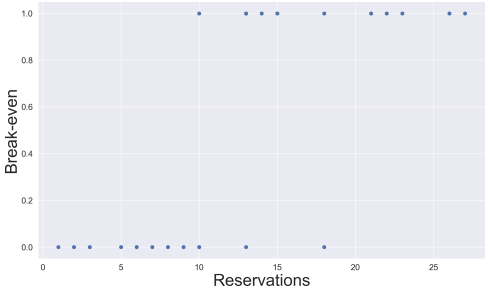
This is a *classification* problem, because we want to classify data as either 0 or 1. We might be tempted to solve this problem with the same code that we used so far. Unfortunately, that approach would fail, and here is why.

Linear regression is all about approximating data with a line, like this:



There's a hidden assumption in linear regression: we assume that the data are roughly aligned to begin with. If the points are arranged on a curve, or scattered around randomly, then we cannot approximate them with a line—and without the line, we cannot make a prediction. The same reasoning applies with multiple input variables: whether we draw a line, a plane, or a higher-dimensional space, we need points that can be reasonably approximated by that shape. Otherwise, we cannot use linear regression.

Now let's take a look at Roberto's file. To make our life easier, let's ignore the "Temperature" and "Tourists" columns for now, and just plot the "Reservations" column against the label:



Even by looking at this one feature, we can see that linear regression wouldn't work well with these points. How are we supposed to draw that line? This is a general problem: linear regression and discrete labels don't mix.

Invasion of the Sigmoids

We found that we cannot crack binary classification with linear regression. However, we don't need to scrap our linear regression code and start from scratch. Instead, we can adapt our existing algorithm to this new problem.

Let's start by looking back at $\hat{y}$, the *weighted sum* of the inputs that we introduced in Adding More Dimensions, on page 48:

```
ŷ = x1 * w1 + x2 * w2 + x3 * w3 + …
```

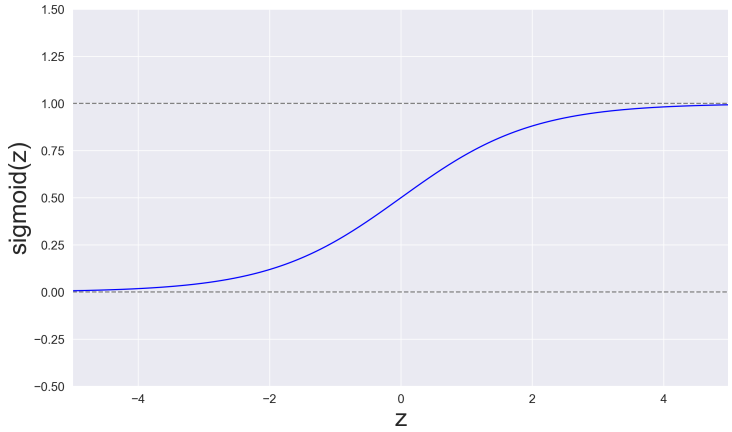
In linear regression, $\hat{y}$ could take any value. Binary classification, however, imposes a tight constraint: $\hat{y}$ must not drop below 0, nor raise above 1. Here's an idea: maybe we can find a function that wraps around the weighted sum, and constrains it to the range from 0 to 1—like this:

```
ŷ = wrapper_function(x1 * w1 + x2 * w2 + x3 * w3 + …)
```

Let me repeat what the `wrapper_function()` does. It takes whatever comes out of the weighted sum—that is, any number—and squashes it into the range from 0 to 1.

We have one more requirement: the function that we're looking for must work well with gradient descent. Think about it: we use this function to calculate $\hat{y}$, then we use $\hat{y}$ to calculate the loss, and finally we descend the loss with gradient descent. For the sake of gradient descent, the wrapper function should be smooth, without flat areas (where the gradient drops to zero) or sudden vertical steps (where the gradient isn't even defined).

To wrap it up, we want a function that smoothly changes across the range from 0 to 1, without ever jumping or going completely flat. Something like this:



As it happens, this is a well-known function that we can use. It's called the *logistic function*, and it belongs to a family of S-shaped functions called *sigmoids*. Actually, since "logistic function" is a mouthful, people usually just call it the "sigmoid," for short. Here is the sigmoid's formula:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The greek letter sigma (σ) stands for "sigmoid." I also used the letter z for the sigmoid's input, to avoid confusion with the system's inputs x.

The formula of the sigmoid is hard to grok intuitively, but its picture tells us everything that we need to know. When its input is 0, the sigmoid returns 0.5. Then it quickly and smoothly falls towards 0 for negative numbers, and raises towards 1 for positive numbers—but it never quite reaches those two extremes. In other words, the sigmoid squeezes any value to a narrow band ranging from 0 to 1, it doesn't have any steep cliffs, and it never goes completely flat. That's the function we need!

Let's return to the code and apply this newfound knowledge.

Confidence and Doubt

Here's a Python version of the sigmoid:

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

We can use sigmoid() to replace our prediction code:

```
def forward(X, w):
    weighted_sum = np.matmul(X, w)
    return sigmoid(weighted_sum)
```

The function above used to be called predict(), and it was just a weighted sum of the inputs. The new version also passes that weighted sum through the sigmoid. Later in this book, we'll see that this process of moving data through the system is also called *forward propagation*—that's why the new function is called forward().

The result of forward() is our prediction $\hat{y}$, that is a matrix with the same dimensions as the weighted sum: one row per example, and one column per weight. Only, each element in the matrix is now constrained between 0 and 1.

Intuitively, you can see the values of y_hat as forecasts that can be more or less certain. If a value is close to the extremes, like 0.01 or 0.98, that's a highly confident forecast. If it's close to the middle, like 0.51, that's a very uncertain forecast.

During the training phase, that gradual variation in confidence is just what we need. We want the loss to change smoothly, so that we can slide over it with gradient descent. Once we switch from the training phase to the classification phase, however, we don't want the system to beat around the bush anymore. The labels that we use to train the classifier are either 0 or 1, so the classification should also be a straight 0 or 1. To get that unambiguous answer, during the classification phase we can round the result to the nearest integer, like this:

```
def classify(X, w):
    return np.round(forward(X, w))
```

The function above could be named predict(), like we used to, or classify(). In the case of a classifier, the two words are pretty much synonyms. I opted for classify() to highlight the fact that we're not doing linear regression anymore, because now we're forecasting a binary value. By the way, statisticians use the word "logistic" to mean "binary," and that's why this technique is called "logistic regression."

It seems that we're making great progress towards a logistic regression program… except for a minor difficulty that we're about to face.

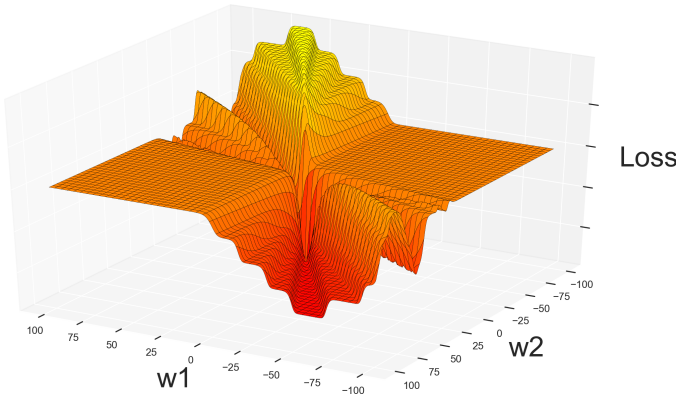
Smoothing It Out

By adding the sigmoid to our program, we introduced a subtle problem: we made gradient descent less reliable. Let's see why.

First, let's update the loss() to use the new classification code:

```
def loss(X, Y, w):
    return np.average((forward(X, w) - Y) ** 2)
```

Because of the sigmoid, this isn't the same loss that we had before. Let's visualize this new loss:



Looks like we have a problem here. See those deep canyons leading straight into holes? We mentioned those holes when we introduced gradient descent—they're the dreaded *local minima*. Remember that the goal of gradient descent is to move downhill? Now consider what happens if GD enters a local minimum: since there is no "downhill" at the bottom of a hole, the algorithm stops there, falsely convinced that it's reached the *global minimum* that it was aiming for.

Here is the conclusion we can take from looking at the diagram above: if we use the mean squared error and the sigmoid together, the resulting loss has an uneven surface littered with local minima. Such a surface is hard to navigate with gradient descent. We'd better look for a different loss function with a smoother, more GD-friendly surface.

We can find one such function in statistics textbooks. It's called the *log loss*. In this case, "log" is short for "logistic," but it's also a reminder that the log loss is based on logarithms:

$$L = -\frac{1}{m}\sum(y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y}))$$

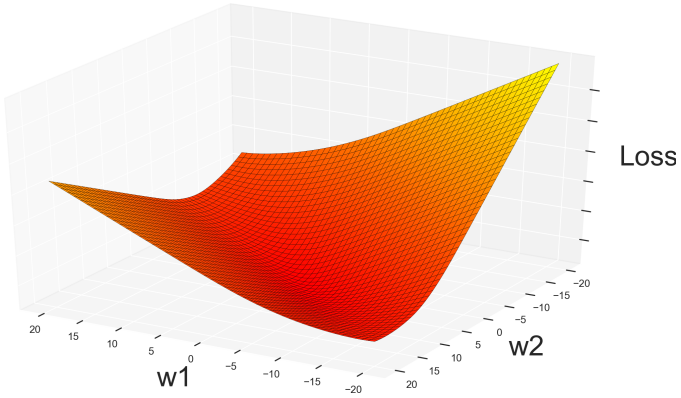
The log loss formula might look daunting, but don't let it intimidate you. Just know that it behaves like a good loss function: the closer $\hat{y}$ is to y, the lower the loss. Also, the formula looks more friendly when turned into code:

```
def loss(X, Y, w):
```

```
    y_hat = forward(X, w)
    first_term = Y * np.log(y_hat)
    second_term = (1 - Y) * np.log(1 - y_hat)
    return -np.average(first_term + second_term)
```

If you're willing to give it a try, you'll find that the log loss is simpler than it looks. Remember that each label in the matrix `Y` is either 0 or 1. For labels that are 0, the `first_term` is multiplied by 0, so it disappears. For labels that are 1, the `second_term` disappears, because it's multiplied by `(1 - Y)`. So each element of `Y` contributes only one of the two terms.

Let's plot the log loss and see what it looks like:



Perfect! No canyons, flat areas, or holes. From now on, this will be our loss function.

Updating the Gradient

Now that we have a brand new loss function, let's look up its gradient. Here is the partial derivative of the log loss with respect to the weight, straight from the math textbooks:

$$\frac{\partial L}{\partial w} = \frac{1}{m} \sum x(\hat{y} - y)$$

If you have good memory, this gradient might look familiar. In fact, it closely resembles the gradient of the mean squared error that we used so far:

$$\frac{\partial MSE}{\partial w} = \frac{1}{m} \sum 2x(\hat{y} - y)$$

See how similar they are? This means that we can take our previous `gradient()` function…

```
def gradient(X, Y, w):
    return 2 * np.matmul(X.T, (predict(X, w) - Y)) / X.shape[0]
```

...and quickly convert it to the new formula:

```
def gradient(X, Y, w):
    return np.matmul(X.T, (forward(X, w) - Y)) / X.shape[0]
```

With this, we're done converting our program from linear regression to logistic regression. Let's run this thing.

Logistic Regression in Action

Here's our final logistic regression code, in all its glory. It loads Roberto's data file, learns from it, and then comes up with a bunch of classifications:

05_discerning/logistic_regression.py

```
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def forward(X, w):
    weighted_sum = np.matmul(X, w)
    return sigmoid(weighted_sum)

def classify(X, w):
    return np.round(forward(X, w))

def loss(X, Y, w):
    y_hat = forward(X, w)
    first_term = Y * np.log(y_hat)
    second_term = (1 - Y) * np.log(1 - y_hat)
    return -np.average(first_term + second_term)

def gradient(X, Y, w):
    return np.matmul(X.T, (forward(X, w) - Y)) / X.shape[0]

def train(X, Y, iterations, lr):
    w = np.zeros((X.shape[1], 1))
    for i in range(iterations):
        print("Iteration %4d => Loss: %.20f" % (i, loss(X, Y, w)))
        w -= gradient(X, Y, w) * lr
    return w

def test(X, Y, w):
    total_examples = X.shape[0]
    correct_results = np.sum(classify(X, w) == Y)
    success_percent = correct_results * 100 / total_examples
    print("\nSuccess: %d/%d (%.2f%%)" %
          (correct_results, total_examples, success_percent))

if __name__ == "__main__":
```

```
# Prepare data
x1, x2, x3, y = np.loadtxt("break_even.txt", skiprows=1, unpack=True)
X = np.column_stack((np.ones(x1.size), x1, x2, x3))
Y = y.reshape(-1, 1)
w = train(X, Y, iterations=10000, lr=0.001)

# Test it
test(X, Y, w)
```

train() didn't change since linear regression, but the other functions did. There is a new sigmoid() function. The old predict() split into two separate functions: forward(), that's used during training, and classify(), that's used for classification. loss() and gradient() changed from the mean squared error to the log loss.

While some code changed since linear regression, the core concepts didn't. We still have a loss() function that tells us how wrong we are, we still have a train() function that finds a matrix of weights by descending the loss' gradient, and we still use those weights in classify() (formerly predict()) to make a classification.

I also added a new test() function that prints the percentage of correct classifications. The instruction np.sum(classify(X, w) == Y) means: first, compare the predicted values to the labels, returning an array that contains True for the elements that match, and False for those that don't; then, count the True elements.

And here is what happens when we run the program with 10000 training iterations:

```
Iteration    0 => Loss: 0.69314718055994495316
Iteration    1 => Loss: 0.68250692927994149883
…
Iteration 9999 => Loss: 0.36572874687292933338

Success: 25/30 (83.33%)
```

The program nailed 25 examples over 30. Not too shabby!

What You Just Learned

In this chapter we walked through *binary classification*: learning and forecasting data that has a binary label. Linear regression doesn't work well with discrete values, so we introduced a new algorithm, *logistic regression*, to deal with classification problems.

Logistic regression is founded on a function called the *logistic function*, or the *sigmoid* for friends. During training, we used the sigmoid to squeeze any value

into the range from 0 to 1. During prediction, we clipped the sigmoid to its nearest binary value—either 0 or 1—to deliver an unambiguous classification.

In linear regression, we used the mean squared error to calculate the loss. Once you add the sigmoid to the recipe, however, the mean squared error's surface becomes bumpy, and hostile to gradient descent. So we switched to an alternative loss function that works well with the sigmoid, called the *log loss*. With that, we can leave linear regression behind for good. It will be all logistic regression from now on.

The next chapter will be a breakthrough. We're about to apply logistic regression to an exciting real-world problem: image recognition. Will our tiny program still work when confronted with *that*?

Hands On: Weighty Decisions

Change the logistic regression program to print the weights after training. You'll see four weights: the first weight is the bias, and the others map to the input variables in Roberto's examples. You'll see that some weights are larger than others, and some might even be negative.

What do those numbers tell us? Which of the columns in Roberto's file has the biggest impact on break even?

CHAPTER 6

Getting Real

It's time to make the leap from simple pizza-related examples to the cool stuff: image recognition. We're about to do something magical. Within a few pages, we'll have a program that recognizes images.

In this chapter and the next, we'll apply our binary classifier to MNIST, a database of handwritten digits. Just a few years ago, before ML systems could tackle more complex datasets, AI researchers used MNIST as a benchmark for their algorithms. In this chapter, we'll join that esteemed crowd.

"Not MNIST Again!"

Besides being a common benchmark in the industry, MNIST is a bit like the "Hello, world" of machine-learning tutorials. In fact, if you had previous exposure to machine learning, you might roll your eyes at the sight of yet another example based on MNIST. Why couldn't I find a more original dataset for this chapter?

Truth is, I picked MNIST *because* it's common, not in spite of that. While most tutorials crack computer vision with the help of high-level ML libraries, we're going to discuss each and every line of code—so we'd better make things easier for ourselves and use a simple, well-documented dataset. Also, a well-known dataset makes it easier for you to look up and compare alternative examples on the Internet.

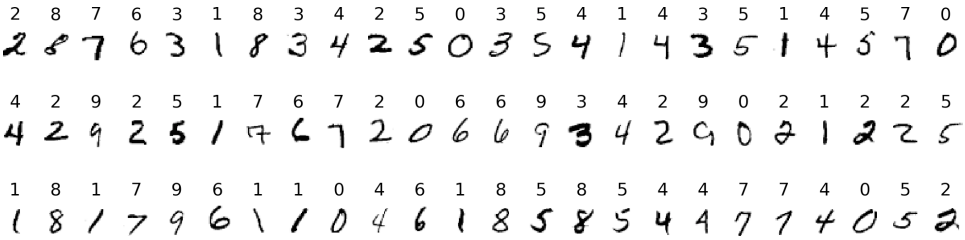
This will be our first experience with computer vision, so let's start small. We'll begin by recognizing a single MNIST digit, and leave more general character recognition to the next chapter.

Data Comes First

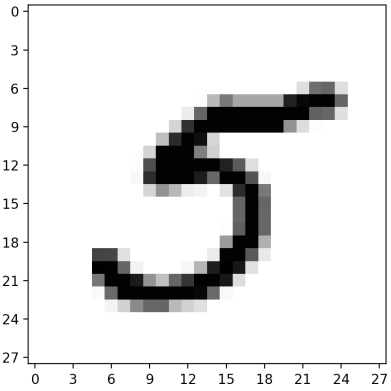
Before we feed data to our ML system, let's get up close and personal with that data. This section tells you all you need to know about MNIST.

Getting to Know MNIST

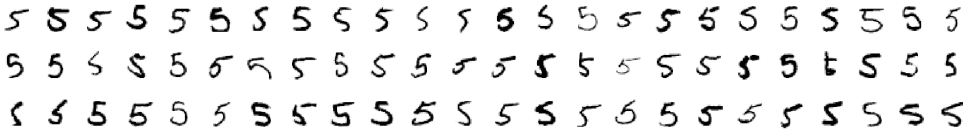
MNIST[1] is a collection of labeled images that's been assembled specifically for supervised learning. Its name stands for "Modified NIST," because it's a remix of earlier data from the National Institute of Standards and Technology. MNIST contains images of handwritten digits, labeled with their numerical values. Here are a few random images, capped by their labels:



Digits are made up of 28 by 28 grayscale pixels, each represented by one byte. In MNIST's grayscale, 0 stands for "perfect background white," and 255 stands for "perfect foreground black." Here's one digit close up:



MNIST isn't huge by modern machine learning standards, but it isn't tiny either. It contains 70,000 examples, neatly partitioned into 7,000 examples for each digit from 0 to 9. Digits have been collected from the wild, so they're quite diverse, as this random assortment of 5s proves:



1. yann.lecun.com/exdb/mnist

Chalk it up to my age, but I have trouble reading some of these 5s myself. Some look like barely-readeable squiggles. If we can write a computer program that recognizes these digits, then we definitely win bragging rights.

Datasets such as MNIST are a godsend to the machine learning community. As cool as ML is, it involves a lot of grindwork to collect, label, clean, and pre-process data. The maintainers of MNIST did that work for us. They collected the images, labeled them, scaled them to the same size, rotated them, centered them, and converted them to the same scale of greys. You can take these digits and feed them straight to a learning program.

MNIST stores images and labels in separate files. There are plenty of libraries that read these data, but the format is simple enough that it makes sense to code our own reader. We'll do that in a few pages.

By the way, you don't have to download the MNIST files: you'll find them amonst the book's source code, in the `data` directory. Take a peek, and you'll see that MNIST is made up of four files. Two of them contain 60,000 images and their matching labels, respectively. The other two files contain the remaining 10,000 images and labels, that have been reserved for testing.

You might wonder why we don't use the same images and labels for both training and testing. The answer to that question requires a short aside.

Training vs. Testing

Consider how we have tested our learning programs so far. First, we trained them on labeled examples. Then we used those same examples to predict values, and we compared the predicted values to the original labels. The closer the two arrays, the better the forecast. That approach served us well as we learned the basics, but it would fail in a real-life project. Here is why.

To make my point, let me come up with an extreme example. Imagine that you're at the pub, chattering about MNIST with a friend. After a few beers, your friend proposes a bet: she will code a system that learns MNIST images in a single iteration of training, and then comes up with 100% accurate classifications. How ludicrous! That sounds like an easy win, but your cunning friend can win such a bet easily. Can you imagine how?

Your friend writes a `train()` function that does nothing, except for storing training examples in a dictionary—a data structure that matches keys to values. (Depending on your coding background, you might call it a "map", a "hashtable", or some other similar name.) In this dictionary, images are keys, and labels are values. Later on, when you ask it to classify an image, the

program just looks up that image in the dictionary and returns the matching label. Hey presto, perfectly accurate forecasts in one iteration of training—and without even bothering to implement a machine learning algorithm!

As you pay that beer, you'd be right to grumble that your friend is cheating. Her system didn't really *learn*—it just *memorized* the images and their labels. Confronted with an image that it hasn't seen before, such a system would respond with an awkward silence. Unlike a proper learning system, it wouldn't be able to generalize its knowledge to new data. Here's a twist: even if nobody is looking to cheat, many ML systems have a built-in tendency to memorize training data. This is a phenomenon known as *overfitting*. You can see that a system is overfitting when it performs better on familiar data than it performs on new data. Such a system could be very accurate when shown images that it's already seen during training, and then disappoint you when confronted with unfamiliar images.

We'll talk a lot about overfitting in the rest of this book. For now, a simple recommendation: *never test a system with the same data that you used to train it*. Otherwise, you might get an unfairly optimistic result, because of overfitting. Before you train the system, set aside a few of your examples for testing, and don't touch them until the training is done.

Now we know how MNIST is organized, and why it's split into separate training and testing set. That's all the knowledge we need. Let's write code that loads MNIST data, and massages that data into a suitable format for our learning program.

Our Own MNIST Library

Let's recap where we are and where we want to go. In the previous chapters, we built a logistic regression program. Now we want to apply that program to MNIST.

As a first step, we need to reshape MNIST's images and labels into an input for our program. Let's see how to do that.

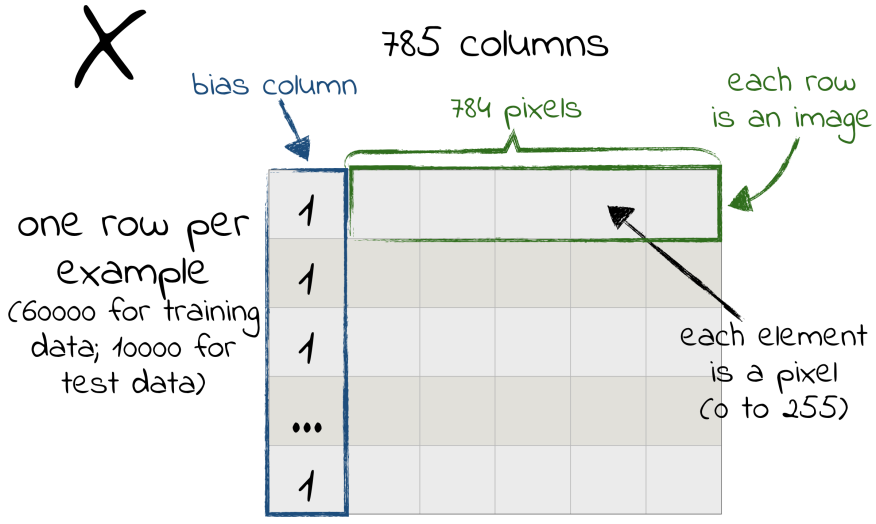
Preparing the Input Matrices

Our logistic regression program expects its input formatted as two matrices: a set of examples `X`, and a set of labels `Y`. Let's start with the matrix of examples `X`.

`X` is supposed to have one line per example and one column per input variable, plus a *bias column* full of 1s. (Remember the bias column? We talked about it in Bye Bye, Bias, on page 60).

To fit MNIST's images to this format, we can reshape each image to a single line of pixels, so that each pixel becomes an input variable. MNIST images are 28 by 28 pixels, so squashing them results in lines of 784 elements. Throw in the bias column, and that makes 785.

So that's what X should look like: a matrix of 60,000 lines (the number of examples) and 785 columns (one per pixel, plus the bias). We just graduated from toy examples with three or four input variables to tens of thousands of examples and hundreds of input variables!



Leap of Faith

In Preparing the Input Matrices, on page 78, we squash each MNIST image into a row of the X matrix. You might have been scratching your head at this idea. Aren't we destroying the images by flattening them? What's the point of having handwritten digits if we grind them into meaningless lines of pixels?

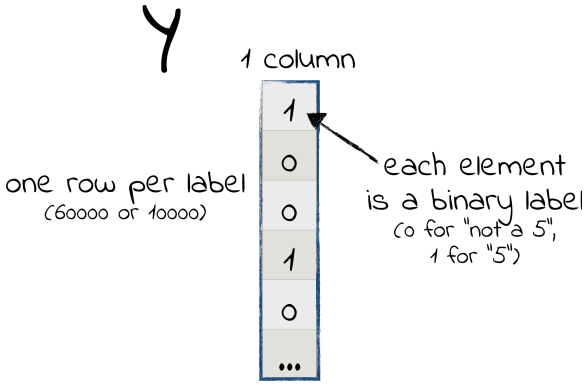
Indeed, we're performing a leap of faith here: we're trusting in the power of statistics. Even though the geometry of those digits is lost, we're betting that the distribution of their pixels is enough information to identify them. For example, the central pixels in a 7 are likely to be darker than the central pixels in a 0.

That being said, later in this book we'll look at more powerful image recognition algorithms, and those algorithms do keep the images' geometry into account.

Now let's look at Y, the matrix of labels. At first glance, it looks simpler than the matrix of images: it still has one line per example, but only one column, that contains the label. However, we have an additional difficulty here: logistic

regression expects a binary label, while MNIST labels range from 0 to 9. How can we fit ten different values into either 0 or 1?

For now, we can work around that problem by narrowing our scope: let's start by recognizing one specific digit—say, the digit 5. This is a problem of binary classification, because a digit can belong to two classes: "not a 5," and "5." This means that we should convert all MNIST labels to 0s ("false"), except for 5s, that we should convert to 1s ("true"). So that's what our `Y` matrix will look like:



Now we know how to build `X` and `Y`. Let's turn this plan into code.

Cooking Data

The Internet overflows with libraries and code snippets that read MNIST data. But we're developers, so hey, let's write one more! In this section we'll code a tiny library that loads those images and labels, and reshapes them into the `X` and `Y` that we've just described.

Loading Images

Here's the code that loads MNIST images into `X`:

06_real/mnist.py

```
import numpy as np
import gzip
import struct

def load_images(filename):
    # Open and unzip the file of images:
    with gzip.open(filename, 'rb') as f:
        # Read the header information into a bunch of variables:
        _ignored, n_images, columns, rows = struct.unpack('>IIII', f.read(16))
        # Read all the pixels into a NumPy array:
        all_pixels = np.frombuffer(f.read(), dtype=np.uint8)
```

```
        # Reshape the pixels into a matrix where each line is an image:
        return all_pixels.reshape(n_images, columns * rows)

def prepend_bias(X):
    # Insert a column of 1s in the position 0 of X.
    # ("axis=1" stands for: "insert a column, not a row")
    return np.insert(X, 0, 1, axis=1)

# 60000 images, each 785 elements (1 bias + 28 * 28 pixels)
X_train = prepend_bias(load_images("../data/mnist/train-images-idx3-ubyte.gz"))

# 10000 images, each 785 elements, with the same structure as X_train
X_test = prepend_bias(load_images("../data/mnist/t10k-images-idx3-ubyte.gz"))
```

load_images() unzips and decodes images from MNIST's binary files. This code is specific to MNIST's binary format, so I wouldn't bother to study each line in detail. (But I wrote detailed comments, in case you want to). The important detail is the shape of the returned matrix: either (60000, 784) or (10000, 784), depending on whether we're loading the training or the test images. Each row in the matrix is a flattened image.

The matrix returned by load_images() doesn't have a bias column yet. To add a bias column, we can pass it to prepend_bias().

Finally, the last few lines of the code above store the training and test images into two constants. The idea is that the client of this library doesn't need to call load_images() and prepend_bias(). Instead, it can import the library (with import mnist) and then refer to these constants (with mnist.X_train and mnist.X_test).

And that's it about the images. Now, the labels.

Loading Labels

This code loads and prepares MNIST's labels:

06_real/mnist.py

```
def load_labels(filename):
    # Open and unzip the file of images:
    with gzip.open(filename, 'rb') as f:
        # Skip the header bytes:
        f.read(8)
        # Read all the labels into a list:
        all_labels = f.read()
        # Reshape the list of labels into a one-column matrix:
        return np.frombuffer(all_labels, dtype=np.uint8).reshape(-1, 1)

def encode_fives(Y):
    encoded_Y = np.zeros_like(Y)
    n_labels = Y.shape[0]
    for i in range(n_labels):
```

```
        if Y[i] == 5:
            encoded_Y[i][0] = 1
    return encoded_Y

# 60K labels, each with value 1 if the digit is a five, and 0 otherwise
Y_train = encode_fives(load_labels("../data/mnist/train-labels-idx1-ubyte.gz"))

# 10000 labels, with the same encoding as Y_train
Y_test = encode_fives(load_labels("../data/mnist/t10k-labels-idx1-ubyte.gz"))
```

load_labels() loads MNIST labels into a NumPy array, and then molds that array into a one-column matrix. Once again, you don't have to understand this code, as you're not likely to load MNIST labels that often—but read the comments if you're curious. (A reminder: reshape(-1, 1) means: "Arrange these data into a matrix with one column, and however many rows you need".) The function returns a matrix with shape (60000, 1) or (10000, 1), depending on whether we're loading the training labels or the test labels.

The matrix returned by load_labels() contains labels from 0 to 9. We can pass that matrix to encode_fives() to turn the labels into binary values. This function prepares a matrix of 0s with the same shape as the one it receives. Then it flips to 1 those elements that match the label 5 in the original matrix. We're not going to live with this code for long, so I didn't bother parameterizing this function. I just hard-coded it to encode the digit 5. The result is a new matrix that contains 1 where the original matrix contained a 5, and 0 elsewhere.

Finally, we have a couple of constants that we can use from our ML code. They store the training labels and the test labels, respectively.

With that, our MNIST library is complete. Let's save it as a file (mnist.py), and use it to feed our ML program.

The Real Thing

It's time to reach for our logistic regression code (from Logistic Regression in Action, on page 72), and run it on MNIST. We do have to adapt it a bit, but the changes are minimal. Indeed, we can use the exact same code that we used in the previous chapter, as long as we update the main code:

06_real/character_recognition.py

```
if __name__ == "__main__":
    import mnist as data
    w = train(data.X_train, data.Y_train, iterations=100, lr=1e-5)
    test(data.X_test, data.Y_test, w)
```

One line for training, one for testing. We don't need to load and prepare the data, because our MNIST library library already takes care of that.

Hyperparameter Blues

While writing the "main" code in The Real Thing, on page 82, I had to find values for `iterations` and `lr` that work well for this new task. As usual, that search involved some trial and error: I just tweaked those hyperparameters until I was happy with the result.

In this specific case, I found out that `lr` must be very small. I settled on a value of 1e-5 (0.00001). With a larger `lr`, the program overflows when calculating the sigmoid and the loss. That overflow is a side effect of using exponentials and logarithms, which can easily churn out enormous numbers. Also, when I tried a slightly larger `lr`, the program failed to converge—which is a fancy way of saying that gradient descent kept skidding around the minimum loss instead of approaching it.

In a future chapter, we'll talk more about tuning hyperparameters. For now, let's just accept that we have to fumble with them whenever we change our algorithm or dataset.

And here's the output of our first image recognition program:

```
Iteration    0 => Loss: 0.69314718055994528623
Iteration    1 => Loss: 0.80042530259490185518
Iteration    2 => Loss: 0.60370180008019158624
...
Iteration   99 => Loss: 0.11895658384798543650

Success: 9637/10000 (96.37%)
```

Over 96% of the program's forecasts proved accurate. Our program recognizes images!

Now that we've basked in the glory of that number for a moment, I have to play the part of the killjoy. 96% looks great, but it doesn't necessarily mean that our program is very accurate. Think about it: only 10% of the digits in the MNIST test set are 5s. This means that a naive program that always forecasts 0 (for: "this is not a 5") would hit the mark 90% of the times. Bummer, I know.

To be fair, our program got nearly 96% correct results, which is better than 90%. However, it's hard to gauge how much better that actually is. To get an intuitive sense of this program's accuracy, we have to extend our code to recognise any digit, from 0 to 9. Let's save our victory dance for the next chapter!

What You Just Learned

In this chapter we got up close and personal with MNIST. We wrote a little library to import MNIST data and reshape it to `X` and `Y` matrices fit for our

logistic regression code. In the end, we used our program to recognize one of the digits in MNIST, with very encouraging results.

Along the way, you learned a few interesting facts about image recognition. You also learned something about testing ML systems, and how the results of a test can be tricky to interpret because of *overfitting*.

In the next chapter, we'll finally tackle the challenge that we set for ourselves at the beginning of this book: recognizing arbitrary digits. How will our code fare?

Hands On: Tricky Digits

Our current code recognizes the digit 5. Which digits would you expect to be easier or harder to recognize than a 5? Make a wild guess.

Now, change the code to recognise one of those other digits. Does the experiment confirm your guesses?

CHAPTER 7

The Final Challenge

In the previous chapter we achieved our goal of building a computer vision system—but only a basic one. The program we built is a *binary classifier*, as it assigns data to one of two classes: "5" and "not 5." Now we're going to push that program further and recognize any digit in MNIST.

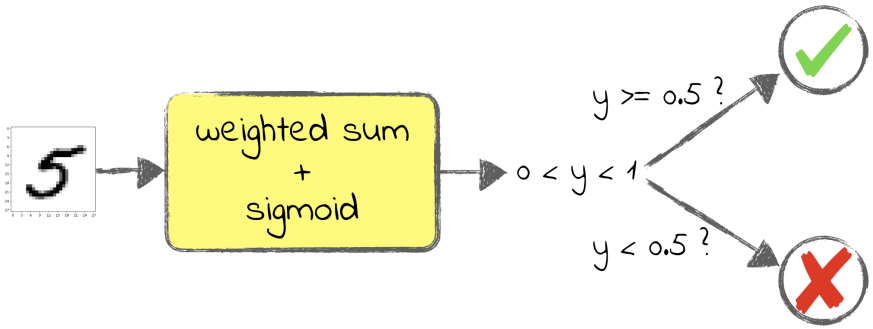
Instead of binary classification, the problem of recognizing digits is called a *multinomial classification* problem, because it involves many classes. For example, a multinomial classifier that recognizes dog breeds would assign data to classes such as "poodle," "beagle," or "border collie."

Don't fret about multinomial classification, because it's well within our grasp. In fact, here's a simple recipe for multinomial classification: build a binary classifier for each class, and then combine those two-classes classifiers into one multi-class classifier.

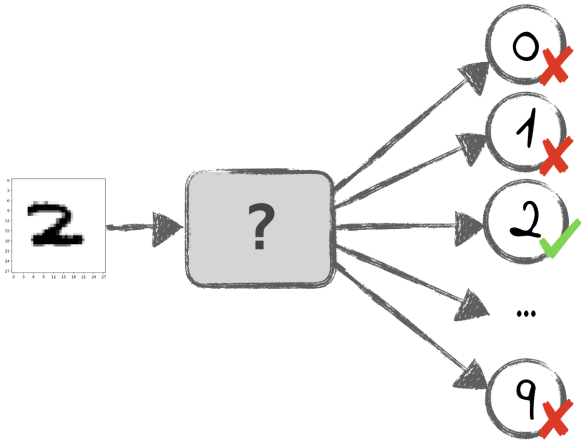
Let's turn that idea into code. Hold on tight: we're tantalizingly close to our goal of building a full-fledged MNIST classifier.

Going Multinomial

Let's recap where we are and where we want to go. We have a binary classifier that's hardwired to recognize a specific digit, such as 5. It passes images through a weighted sum, and then a sigmoid. The result is a number ranging from 0 to 1. Then we round that number to either 1 or 0, because we want a binary result—a straight "yes, this is a 5" or "no, this isn't a 5."



And here is where we want to go: we want a program that takes an image and tells us which digit that image represents, from 0 to 9.



Let's see how we can go from here to there. First, focus on that box right at the center of the first picture—a weighted sum, followed by a sigmoid. There is no common name for this sequence. For short, let's call it WSS, for "Weighted Sum plus Sigmoid." An WSS is just like a binary classifier, minus the last step: instead of returning either 0 or 1, it returns a floating point number between 0 and 1.

Now imagine building an array of ten WSSs, one per class—from the 0-WSS, that only recognizes zeros, to the 9-WSS that... well, you got the idea. If we run them all, then we get an array of ten numbers like this one:

"0"	"1"	"2"	"3"	"4"	"5"	"6"	"7"	"8"	"9"
0.111	0.005	0.787	0.170	0.001	0.176	0.352	0.001	0.073	0.003

These numbers tell us how confident each WSS is. For example, the 2-WSS returns 0.787, which is pretty close to the maximum value 1. This means

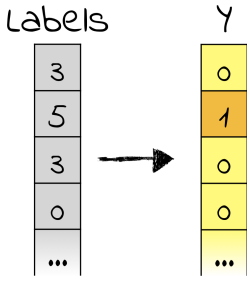
that the 2-WSS is pretty confident that the image is indeed a 2. By contrast, the 4-WSS returns a very low number, so it's pretty sure that the image is *not* a 4. The other WSSs also don't believe that they're looking at their own digit. Overall, the 2-WSS is the most confident of the bunch, so this image is probably a 2.

Now we have a more detailed plan for how to do multinomial classification. First, run ten WSSs on the image, each specialized for a different digit. Second, pick the digit that gets the highest confidence from its matching WSS.

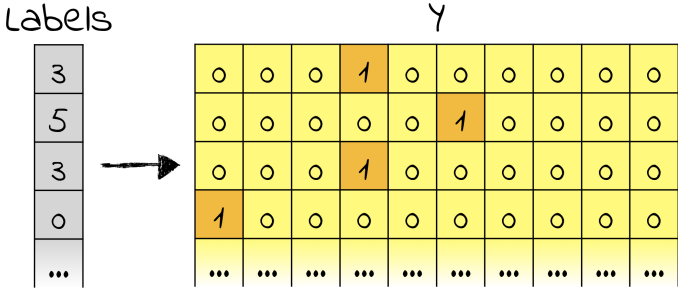
We could implement this plan by running the same WSS code ten times, once per class. We could… but we can do better.

One Hot Encoding

Remember how we encoded labels in Loading Labels, on page 81? Back then, we only cared about telling apart 5 from other digits. So we encoded the labels by replacing 5 with 1, and other digits with 0:



Now that we need to recognize ten digits, we could come up with ten such encoded matrices, one per class. But here is a better way: we can have one big matrix with ten columns, where each column encodes a digit. The result would look like this:



In the example above, the first label is a 3, so the fourth cell in the second row of `Y` is a 1. (It's the fourth cell, not the third, since we're enumerating the

digits starting from 0). Similarly, each row has a 1 in the position matching its original label, and zeros everywhere else.

This way of encoding labels is called *one hot encoding*, because only one value per row is a hot 1. The rest are cold, cold zeros.

Can you see where this is going? We won't run our code ten times, once per class. Instead, we'll run it once, with one column per class. It's as if each columm in the matrix contained the binary encoding for one of the WSSs. Thanks to the power of NumPy, this matrix-based approach works just as well as running a classifier ten times—only faster, and with less code.

Encoding Non-Numerical Labels

One hot encoding isn't just for numerical labels, such as those in MNIST. On the contrary, you can one hot encode any enumerated set of labels. If our labels were, say, "duck," "platypus," and "tapir," then we could sort them in an arbitrary (but fixed) order, and encode them as `[1, 0, 0]`, `[0, 1, 0]`, and `[0, 0, 1]`, respectively. In the case of MNIST, the labels happen to be numbers, so it's convenient to sort them in numerical order—but it doesn't have to be that way.

Think of one hot encoding as a simple dictionary that maps the human-readable label to a fixed-length sequence of zeros containing a single 1.

One hot encoding is redundant: instead of one number per example, we have as many numbers as we have classes. But that redundancy is usually worth it, as we're about to see in the next section.

One Hot Encoding in Action

In Our Own MNIST Library, on page 78, we wrote a library to load and prepare MNIST data. That library encoded the labels with a function called `encode_fives()`. Let's replace that function with a new one:

07_final/mnist.py

```
def one_hot_encode(Y):
    n_labels = Y.shape[0]
    n_classes = 10
    encoded_Y = np.zeros((n_labels, n_classes))
    for i in range(n_labels):
        label = Y[i]
        encoded_Y[i][label] = 1
    return encoded_Y
```

`one_hot_encode()` initializes a matrix of zeros with one row per label, and one column per class. (`Y.shape[0]` means "the number of rows in `Y`"). Then it walks

through the matrix, flipping the "hot" values to 1. We can use this function to initialize a one hot encoded version of Y_train:

```
# 60K labels, each a single digit from 0 to 9
Y_train_unencoded = load_labels("../data/mnist/train-labels-idx1-ubyte.gz")

# 60K labels, each consisting of 10 one-hot encoded elements
Y_train = one_hot_encode(Y_train_unencoded)

# 10000 labels, each a single digit from 0 to 9
Y_test = load_labels("../data/mnist/t10k-labels-idx1-ubyte.gz")
```

One detail in the code above might puzzle you: why do we encode the training data, but not the test data? That's a common source of confusion, so let's clarify that point by looking at the classify() function.

Decoding the Classifier's Answers

Let's review how classify() works. During the classification phase, the WSSs are going to return arrays of ten numbers from 0 to 1. But when we ask the system to recognize an image, we don't want to see those arrays—we want a human-readable answer such as "3." That means that we must decode the WSSs' answers before returning them.

So far, the classify() function didn't have much to do, apart from calling forward() and rounding its output. Now classify() has a more complex job. It must convert the output of the WSSs back to human-readable labels, like this:

07_final/mnist_classifier.py

```
def classify(X, w):
    y_hat = forward(X, w)
    labels = np.argmax(y_hat, axis=1)
    return labels.reshape(-1, 1)
```

The first line of classify() calculates a matrix of predictions, with one row per label, and one column per class. Each row in this matrix contains ten numbers between 0 and 1.

The second line uses NumPy's argmax() function to get the index of the maximum value in each row of y_hat—the value that's closer to 1. By default, argmax() finds the maximum over the entire matrix, so this code uses the argument axis=1 to apply it to each row separately. The result is an array of indices, which are also the decoded MNIST labels. (Functions like argmax() take some getting used to, so you might want to play with them in an interactive Python interpreter before you use them. On the other hand, they make for terse, efficient code.)

Finally, the last line of classify() reshapes the labels array to a single-column matrix of digits. Now we finally see why the Y_test matrix isn't one hot encoded like Y_train. We're going to compare Y_test with the classifier's output, so the two matrices must look the same: a single column of human-readable labels.

We're almost there. We just need one more change to the classifier code.

We Need More Weights

When we introduced one hot encoding, we extended the matrix of labels from one to ten columns. Now we need to do the same with the weights.

So far, our matrix of weights had one column, and one row per input variable. We initialized it like this:

```
w = np.zeros((X.shape[1], 1))
```

Now we need ten columns of weights, one per class:

```
w = np.zeros((X_train.shape[1], Y_train.shape[1]))
```

The new w has one row per input variable, and one column per class. The code above gets the number of input variables and classes from the number of columns in X and Y, respectively.

At this point, you could well be confused by all those matrix dimensions. I have to confess that I had to pause and double check them while writing this section. ("Wait, does w have as many rows as X has columns, or is it the other way around?"). Let's take a moment to get a sanity check about all those rows and columns.

Matrix Dimensions, Reviewed

I have fond memories of my first forays in machine learning—except for one frustrating, error-prone activity: I never seemed to get those darn matrix dimensions rights. After suffering through so many "shapes not aligned" errors, I'd like to spare you that experience. At the risk of being redundant, here's a recap of the dimensions of our matrices.

One note before I dive in: when it comes to matrix dimensions, I find it very useful to sketch them on paper. Try it—it works great for me.

- X is (m, n)—one row per *example*, and one column per *input variable*. The MNIST training set has 60,000 examples, each composed by 784 pixels. Add the bias column, and X becomes (60000, 785).

- `Y` is a matrix of one hot encoded labels. It has one row per *example*, and one column per *class*. If we use `k` for the number of classes, then `Y` is `(m, k)`. In our case, that's `(60000, 10)`.
- The matrix of weights `w` is `(n, k)`—one row per *input variable*, and one column per *class*. In our case, that's `(785, 10)`. We covered a lot of ground since our first learning program, where `w` was a single parameter. Now we're training over thousands of parameters!

As a sanity check, we can check that the weighted sum $X \cdot w = Y$ has all the right dimensions. And in fact, it does:

$$(m, n) \cdot (n, k) = (m, k)$$

(If you can't remember how the dimensions of matrix multiplication work, review Multiplying Matrices, on page 51.)

One last reminder: the numbers above are the sizes of the matrices during training. The MNIST test set is smaller, and so are the matrices. For one, `X` is `(10000, 785)` during testing. If we classify a single image, then `X` is a single-row `(1, 785)` matrix.

After this tedious check through matrix dimensions, you must be eager to review and run the classifier. Let's do it.

Moment of Truth

This is the time we've been waiting for. We're about to unleash our multinomial classifier on MNIST. Here is the classifier's code, in all its glory:

07_final/mnist_classifier.py

```
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def forward(X, w):
    weighted_sum = np.matmul(X, w)
    return sigmoid(weighted_sum)

def classify(X, w):
    y_hat = forward(X, w)
    labels = np.argmax(y_hat, axis=1)
    return labels.reshape(-1, 1)

def loss(X, Y, w):
    y_hat = forward(X, w)
    first_term = Y * np.log(y_hat)
    second_term = (1 - Y) * np.log(1 - y_hat)
```

```
    return -np.sum(first_term + second_term) / X.shape[0]

def gradient(X, Y, w):
    return np.matmul(X.T, (forward(X, w) - Y)) / X.shape[0]

def report(iteration, X_train, Y_train, X_test, Y_test, w):
    matches = np.count_nonzero(classify(X_test, w) == Y_test)
    n_test_examples = Y_test.shape[0]
    matches = matches * 100.0 / n_test_examples
    training_loss = loss(X_train, Y_train, w)
    print("%d - Loss: %.20f, %.2f%%" % (iteration, training_loss, matches))

def train(X_train, Y_train, X_test, Y_test, iterations, lr):
    w = np.zeros((X_train.shape[1], Y_train.shape[1]))
    for i in range(iterations):
        report(i, X_train, Y_train, X_test, Y_test, w)
        w -= gradient(X_train, Y_train, w) * lr
    report(iterations, X_train, Y_train, X_test, Y_test, w)
    return w

if __name__ == "__main__":
    import mnist as data
    w = train(data.X_train, data.Y_train,
              data.X_test, data.Y_test,
              iterations=200, lr=1e-5)
```

I took this chance to extract a new report() function that logs the percentage of correct results. report() is similar to the test() function that we had before, but it's called once per training iteration, plus once at the very end. This log shows us in detail how well (or badly) our classifier is learning.

A Tiny Fix

You'd have to be eagle-eyed to spot it, but the final classifier code in Moment of Truth, on page 91 introduces a subtle change into the last line of loss(). So far, that line used NumPy's average() function to calculate the average of (first_term + second_term). However, average() calculates the average over all the *elements* of a matrix, while we want the loss over the examples—which means all the *lines* in the matrix.

So far, that detail didn't matter, because (first_term + second_term) had exactly one element per line. Since we switched to one hot encoding, however, that matrix has 10 elements per line—so the original code would calculate a loss that's 10 times smaller than the correct one. The new code fixes that problem by calculating the average in the old-fashioned way: it sums all the elements in the matrix, and then divides by the number of lines.

That bug is further proof that working with matrices can trip you up. At least, it tripped me: that mistake almost slipped through this book's code reviews.

And here's what happens when the program runs:

```
0 - Loss: 6.93147180559945397249, 9.80%
1 - Loss: 8.43445687508333641347, 68.04%
2 - Loss: 5.51204748892387641490, 68.10%
3 - Loss: 2.95687007359365416903, 68.62%
…
200 - Loss: 0.85863196488041293453, 90.32%
```

Let's take a deep breath, because we just built something wonderful. In a few minutes of number-crunching, our little program learned to recognize handwritten digits with over 90% accuracy.

Whitespace excluded, the entire program spans some 35 lines of Python. We could easily squeeze it down to 20 lines if we were willing by sacrificing some readability and logging. Such a short program, and we didn't even use any ML library. There is no rabbit hole of complex code hiding amongst those lines. We can literally understand the entire thing.

What You Just Learned

Let's recap our first adventure in machine learning:

- In Chapter 1, How Machine Learning Works, on page 3 we learned what machine learning and *supervised learning* are.
- In Chapter 2, Your First Learning Program, on page 15 we got our first concrete taste of supervised learning: we used *linear regression* to predict one variable from another.
- In Chapter 3, Walking the Gradient, on page 33 we upgraded the learning program with a faster and more efficient algorithm: *gradient descent*.
- In Chapter 4, Hyperspace!, on page 47, we took advantage of gradient descent (and a bit of matrix magic) to implement *multiple linear regression*—like linear regression, only with multiple inputs.
- In Chapter 5, A Discerning Machine, on page 65 we leaped from multiple linear regression to *logistic regression* and *classification*.
- In Chapter 6, Getting Real, on page 75 we used our binary classifier to recognize a single digit in the *MNIST* dataset.
- Finally, in this chapter we bumped up to *multinomial classification*, recognizing all MNIST characters with over 90% accuracy.

What a journey! Let's have one last exercise. Then, in the next chapter, we'll stop coding for a while and take a bird's-eye view of this thing we've built.

Hands On: The CIFAR Challenge

If you're up for a challenge, here's an optional exercise for you: run mnist_classifier.py on the CIFAR-10 dataset[1]. CIFAR-10 is like MNIST on steroids: it contains images from 10 classes of subjects, including cats, airplanes, and birds. We'll talk more about CIFAR-10 in Part III of this book. For now, you'll have to read the documentation on the CIFAR-10 web page and write your own code to load and prepare the data.

Don't underestimate this little project. If you're not steeped in Python, it might easily take you a few hours to write the CIFAR equivalent of the mnist.py library. If you'd rather move forward, there is nothing shameful in that. We'll come back to CIFAR later, armed with more powerful algorithms.

Should you accept the challenge, here are a few hints for you:

- Apart from hyperparameters, you shouldn't need to change any code in the classifier itself. All changes should go into the code that loads and prepares data.
- CIFAR-10 images have red, green, and blue channels. Treat the channels as if you had three times as many pixels, flattening them into one long row.
- Remember to add a bias column to both X_train and X_test.
- Remember to one hot encode Y_train, but not Y_test.
- Use a small learning rate during training, such as 1e-6. Large learning rates cause errors when calculating the loss, because the logarithms and exponentials generate huge (or tiny) numbers.

Don't expect results on par with those we achieved on MNIST. Our current system is too limited to recognize cats or airplanes accurately.

If you get stuck, check my solution in the book's source code. It's in the final directory.

1. https://www.cs.toronto.edu/~kriz/cifar.html

CHAPTER 8

The Perceptron

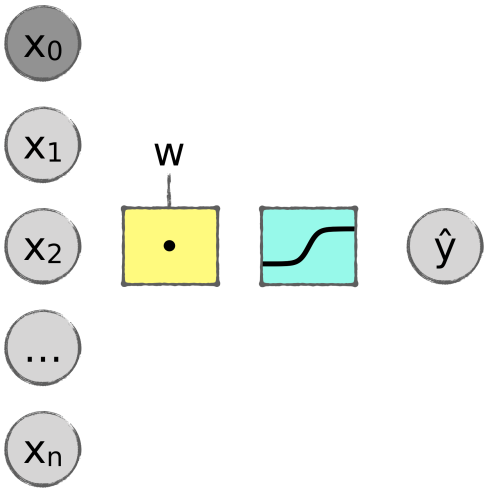
In the previous chapters we toiled through a lot of nitty-gritties. It's time to take a step back and enjoy the big picture. This chapter takes a broad view of the system we've built—a system that ML pundits would call a *perceptron*.

At the end of this chapter, you'll know what a perceptron looks like and what it can do. Crucially, you will also know what it can *not* do, and why we must move forward into more sophisticated algorithms such as neural networks.

Besides wrapping our heads around the perceptron, we're also going to talk about its history. Far from being a boring history lesson, this will be an epic war tale—a clash of ideas that impacted much of what we know about computers.

Enter the Perceptron

To understand what a perceptron is, let's take a look back at the binary classifier we wrote in Chapter 6, Getting Real, on page 75. That program sorted MNIST characters into two classes: "5" and "not a 5." Here's one way to look at it:



The diagram above tracks an MNIST image through the system. The process begins with the *input variables*, from x_1 to x_n. In the case of MNIST, the input variables are the 784 pixels of an image. To those, we added a bias x_0, with a constant value of 1. I colored it a darker shade of grey, to make it stand apart from other input variables.

The next step is the *weighted sum* of the input variables. It's implemented as a multiplication of matrices, so it's stamped with the "dot" sign.

The weighted sum flows through one last function. In general, this is called the *activation function*. Different learning systems use different activation functions. In our case, we used a sigmoid. The result of the sigmoid is the output $\hat{y}$, ranging from 0 to 1.

During training, the system compares $\hat{y}$ with the labels to calculate the next step of gradient descent. During classification, it snaps the value of $\hat{y}$ to one of its extremes—either 1 or 0, meaning "5" and "not a 5," respectively.

The architecture I just described is the *perceptron*, the most basic learning network.

The Textbook Perceptron

If you look it up on Wikipedia, you'll see that the textbook perceptron is actually simpler than the variant I discuss in this chapter. Instead of gradient descent, the perceptron uses a simpler algorithm, similar to the one we implemented in Chapter 2, Your First Learning Program, on page 15.

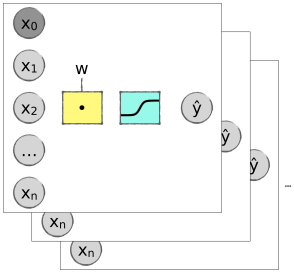
Since it doesn't use gradient descent, the vanilla perceptron doesn't need an activation function with a smooth gradient such as the sigmoid. Instead, it can get away with

a simple “step” function that snaps the value of the weighted sum to either 1 or 0, depending on whether it’s positive or negative.

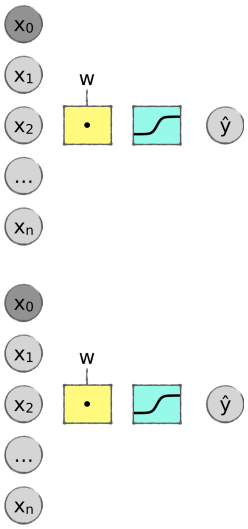
Assembling Perceptrons

The perceptron is a great building block for more complex systems. In a sense, we have been assembling multiple perceptrons since the very beginning. Let’s see how.

During training, our system reads all the examples together, rather than one example at a time. In a way, that operation is like “stacking” multiple perceptrons, sending one example to each perceptron, and then collecting all the outputs into a matrix:



In Chapter 7, The Final Challenge, on page 85, we also assembled perceptrons in a different way. A perceptron is a binary classifier—it classifies things as either 0 or 1. To classify ten digits, we used ten perceptrons, each classifying one digit against all the others. You can think of this as using the ten perceptrons in parallel:



...

Each parallelized perceptron classifies one class, from 0 to 9. During classification, we pick the class that outputs the largest `y`.

So we *stacked* perceptrons, and we *parallelized* perceptrons. In both cases, we did it with matrix operations, which was easier and faster than running the same classifier multiple times—once per example, and then once per class.

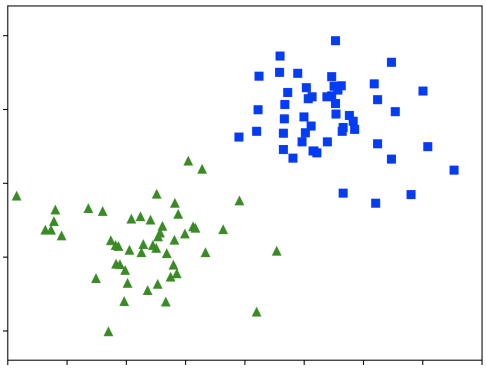
There is one more way to combine perceptrons: *serialize* them, using the output of one perceptron as input to the next. The result is called a *multilayer perceptron*. We didn't use multilayer perceptrons yet, but we will… a lot. For now, just keep this idea at the back of your mind.

Where Perceptrons Fail

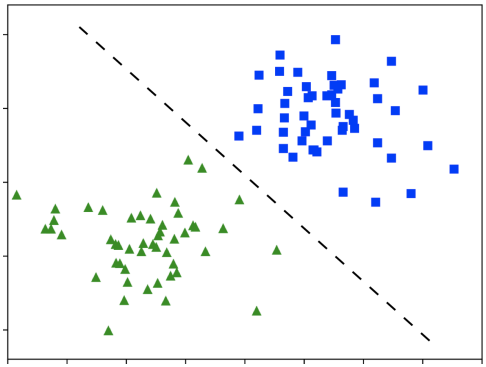
What's not to love about perceptrons? They're simple, and they can be assembled into larger structure like machine learning construction bricks. However, that simplicity comes with a distressing limitation: perceptrons work well on some kind of data, and fail badly on others. Let's look at a few examples.

Good Data…

Take a look at this two-dimensional dataset:



The dataset above contains two classes of data—green triangles and blue squares arranged into neatly distinct clusters. You could even separate the two classes with a line, like this:



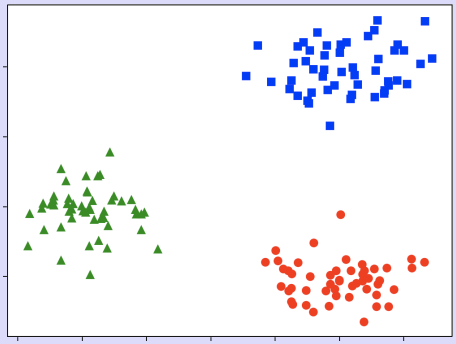
Datasets like the one above, that can be partitioned with a straight line, are called *linearly separable* datasets. If a dataset can only be partitioned with a curved line, or cannot be partitioned at all, then it's not linearly separable.

Perceptrons are good at classifying linearly separable data. Later in this book, in Chapter 12, How Classifiers Work, on page 147, we'll see why that's the case. For now, you can take it as a given: train a perceptron on a linearly separable dataset, like the one above, and you'll get accurate classifications.

We just saw an example with two-dimensional data. However, the same reasoning applies for data with three or more dimensions—only, instead of a line, you'd have to separate the classes with a higher-dimensional linear shape. With three dimensions, you'd have to trace a plane; with 785 dimensions, as in the case of MNIST, you'd have to trace a 784-dimensional linear shape. (Good luck with that!) In all of these cases, the intuition stays the same: if you can separate the classes with a "straight" shape, then you're golden: those data are good for a perceptron.

Adding More Classes

In Good Data..., on page 98, we use a binary classifier to show the meaning of "linearly separable." But what if a dataset has more than two classes, as in the case below?



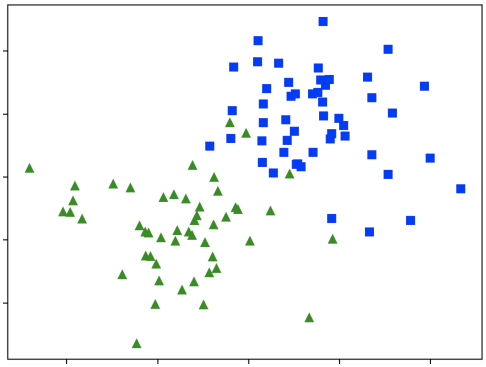
With three classes, we need a *multinomial* (rather than *binary*) classifier, like the one that we built to recognize MNIST digits. However, you might remember that we built that multinomial classifier by combining multiple binary classifiers. We could do the same to classify the dataset above: to recognize those three classes, we'd have three binary classifiers, one per class.

Long story short: even with more than two classes, the concept of linear separability stays pretty much the same. In particular, the data above is linearly separable, because you could trace three straight lines that separates each class from all the other classes. As a result, a perceptron-based classifier would be cool with this dataset.

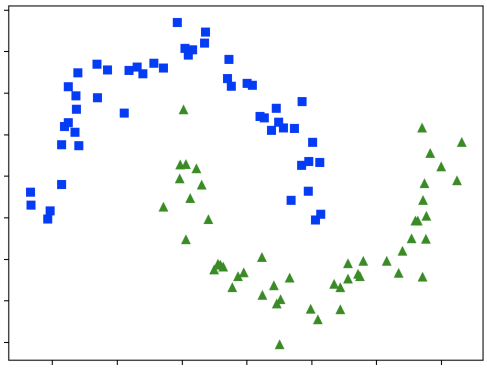
Now let's look at data that's *not* perceptron-friendly.

...Bad Data

Look at these data:



The dataset above isn't linearly separable. Whichever line you draw, some data points will fall into the opposite class's camp. In this specific case, a perceptron could still separate most squares from most triangles, and come up with an OK classification. As the data gets more tangled, however, things go south quick, as this last example demonstrates:



There is no way to separate the points above with a line—not even close. However you trace that line, at least one partition will contain an almost-even mix of squares and triangles. A perceptron would be pretty bad at classifying these points.

Granted, most real-life datasets aren't as viciously twisted as the one above. After all, MNIST is not linearly separable, and we still got a cool 91% accuracy on it. Think about it for a moment, however, and you'll realize that 91% is not that good in practice. One mistake every ten characters is a pretty high error rate for most useful applications.

To make matters worse, that 91% on MNIST is about as far as our perceptron can go. If you tried the CIFAR challenge at the end of the previous chapter, then you know that more complex data are beyond the perceptron's reach. There is no way around this: to write useful ML software, we need more than stacked and parallelized perceptrons.

So, that's the perceptron in a nutshell: it's simple, and it only works well on linearly separable data. That's a well understood trade-off today, but that wasn't always the case. In fact, the perceptron was once the center of a grand war of ideas—one that changed the history of computing as we know it. Bear with me for a couple of pages, because that's a story worth telling.

A Tale of Perceptrons

In the 1950s, the nascent field of Artificial Intelligence was split into rivaling factions. Two groups competed for academic mindshare: the top dogs were

the "symbolists," lead by authorities such as Marvin Minsky and John McCarthy; the runners-up were the "connectionists," led by the charismatic Frank Rosenblatt.

The two tribes had very different approaches. The symbolists believed in *programming* an intelligent machine from the ground up. Piece by piece, they planned to build computers that would eventually manipulate concepts faster and better than humans.

This idea might sound overly optimistic today, but it made perfect sense in the 1950s, when people were inventing the first high-level programming languages. Those languages seemed much closer to human thinking than plain old assembly language. Who knew how far that process could go? (In fact, John McCarthy invented one of the first high-level programming languages, LISP, in his quest to code intelligence.)

The opposite faction, the connectionists, chased another dream. Their idea could be summed up as: build a brain, and intelligence will come.

To simplify things, the brain is made of neurons connected through fibers. Each neuron has multiple input fibers, and one output fiber. If the inputs are active in a certain pattern (maybe because they get a signal from the sensory organs), then the output also activates. The connectionist leader Frank Rosenblatt built a machine inspired by that mechanism. By analogy with neurons, he named the machine "perceptron," and its final processing step "activation function."

The first perceptron was a far cry from our tiny Python program. The "Mark 1 perceptron" was a room-sized piece of hardware that looked a bit like a server rack covered by an impenetrable tangle of wires. It had a camera connected to 400 photocells—essentially, very lo-res pixels. The weights were implemented with potentiometers wired to the photocells. During the learning phase, the potentiometers were physically rotated by electric motors.

To overcome the perceptron's limitations, the connectionists also studied *multilayer perceptrons*, which seemed able to tackle non-linearly separable data. Meanwhile, the symbolists were busy writing programs that solved algebra problems and stacked construction blocks with a robot arm.

To be fair, neither faction was making much progress towards intelligent machines. On the other hand, both factions were inclined to ballyhoo and extravagant promises. At one point, Rosenblatt declared that the perceptron was the first step towards machines that would not only be damn smart, but

even self-conscious. The popular press bought into it hook, line and sinker, making symbolists jealous.

The feud went on for years, with the symbolists reaping the lion's share of research funds, and the connectionists playing the part of the popular underdogs. Then, at some point, things really hit the fan.

The Final Battle

From the 50s to the mid-60s, connectionists had been nibbling at AI research funds. The powerful symbolist leader Marvin Minsky thought that was a waste of money, and decided to set things straight once and for all.

Minsky's plan was simple: he would study the connectionist's ideas, with an eye towards showing their limitations. Together with the like-minded Seymour Papert, he published an entire book on the topic, called "Perceptrons." The book was essentially Minsky's way to damn perceptrons with faint praise. It focused a lot on what perceptrons could *not* learn—such as non-linearly separable data.

To their credit, Minsky and Papert admitted that multilayer perceptron could overcome the limitations of regular perceptrons. However, they hasted to add their gut feeling: multilayer perceptrons were probably impossible to train. In their opinion, the whole idea of building intelligence with perceptrons was little more than a pipe dream.

Bolstered by Minsky's reputation, the "Perceptron" book had more impact than the authors themselves intended. Where Minsky and Papert had been nuanced, the scientific community went for the "too long; didn't read" version: "connectionism is a dead end." Within a few months, funding for connectionist research dried up.

The impact of "Perceptrons" didn't stop there. The public had been waiting for a perceptron to stand up and ask for a cup of tea anytime soon. Now one of the major AI eggheads was scoffing at the whole thing. The popular opinion switched, and all of connectionism was filed under "unscientific bollocks".

In the early 70s, Rosenblatt died in a sailboat accident. That seemed like the last nail in the perceptron's coffin.

The Aftermath

After connectionism was disgraced, it almost disappeared from academic research. Only a handful of researchers over the world, like medieval monks, kept the study of perceptrons alive.

Minsky's parting questions loomed over them like a prophecy of doom. Were multi-layer perceptrons really impossible to train? Was the entire idea a dead end? Fifteen years passed before they could answer those questions.

What they found is the subject of the next part of this book.

Part II

Neural Networks

In this second part of the book, we'll use what we learned in Part I to build a more sophisticated learning machine: a neural network. Neural networks are leaps and bounds more powerful than perceptrons, but they come with their own set of challenges. We'll spend a few chapters building our first network, and a few more detailing and overcoming those challenges.

Here's a mild spoiler: even the first version of our neural network will classify MNIST digits better than the perceptron. But we won't stop there! As you'll see soon, we're going to aim for a target that's way more ambitious.

Fasten your seatbelts…this will be quite the trip.

CHAPTER 9

Designing the Network

Part I of this book was all about the perceptron. Part II is about the perceptron's big brother, and the most important idea in this book: the *neural network*. Neural networks are way more powerful than perceptrons. In Where Perceptrons Fail, on page 98, we learned that perceptrons need simple data that is linearly separable. By contrast, neural networks can deal with gnarly data, like photos of real-world objects.

Even on a simple dataset like MNIST, our perceptron was just scraping by, making almost one mistake every ten characters. With neural networks, we can aim for an order of magnitude better accuracy: in this part of the book, we'd like to build an MNIST classifier that comes close to 99% accuracy—one error every 100 characters.

Here's a plan to reach for that lofty goal: over the next few chapters, we'll build and tweak a neural network. Once again,you'll learn by doing, handcrafting code line by line. It will take us three short chapters to complete our neural network v1.0:

- In this first chapter we'll design a network that classifies MNIST digits—on paper.
- In the next chapter, Chapter 10, Building the Network, on page 117, we'll write the network's *classification* code.
- In Chapter 11, Training the Network, on page 127, we'll write the network's *training* code.

After these three chapters, we'll have a working neural network. Even then, however, you might struggle to understand how it works. The following chapter, Chapter 12, How Classifiers Work, on page 147, will use our network as a concrete example to answer questions like: "What makes neural networks so powerful?"

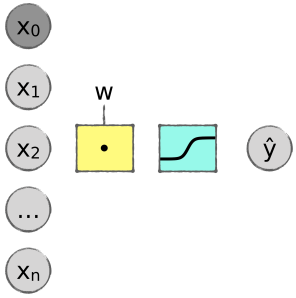
Even after building a neural network and understanding how it works, we'll have some more work to do. The first version of our network won't be very accurate—in fact, it will only be about as accurate as the perceptron. In the remaining three chapters of Part II, we'll unleash the network's power and aim straight for that 99% accuracy.

I won't spoil your fun by telling you whether we'll eventually hit our goal. To find out, you'll have to keep reading. (Or peek at the last pages of Part II… But you wouldn't do that, would you?)

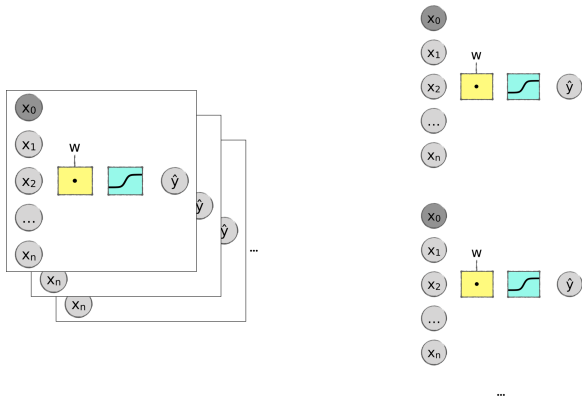
Now that we have a plan for action, let's dive straight into our first task: designing a neural network to classify MNIST digits.

Assembling a Neural Network from Perceptrons

Let's see how to build a neural network, starting with the perceptron that we already have. As a reminder, here is that perceptron again: a weighted sum of the inputs, followed by a sigmoid.



In Part I of this book, we didn't just use the perceptron as-is—we also combined perceptrons in two different ways. First, we trained the perceptron with many MNIST images at once; and second, we used ten perceptrons to classify the ten possible digits. In Assembling Perceptrons, on page 97 we compared those two operations to "stacking" and "parallelizing" perceptrons, respectively:

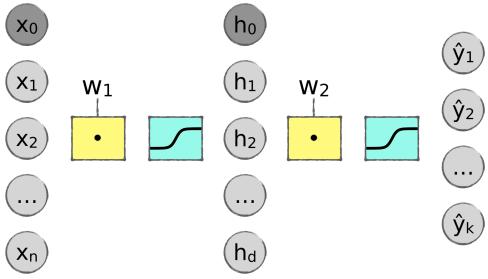


To be clear, we didn't *literally* stack and parallelize perceptrons. Instead, we used matrices to get a similar result. Our perceptron's input was a matrix with one row per image; and our perceptron's output was a matrix with ten columns, one per class. The "stacking" and "parallelizing" metaphors are just convenient shortcuts to describe those matrix-based calculations.

Now we're about to take these extended (stacked and parallelized) perceptrons, and use them as building blocks for a neural network.

Chaining Perceptrons

We can build a neural network by *serializing* two perceptrons like this:



See? Each perceptron has its own weights and it own sigmoid operation, but the outputs of the first perceptron are also the inputs of the second. To avoid confusion, I renamed those values with the letter `h`, and I used `d` to indicate their number. `h` stands for "hidden," because these values are neither part of the network's input, nor the output. (And since I know that you're going to ask: the `d` stands for nothing in particular. I just took a cue from mathematicians and came up with a random letter.)

Back in the day, a structure like the one above was called a *multilayer perceptron*, or an *artificial neural network*. These days, most people simply call it a neural network. The round grey bubbles are called *nodes*, and they're arranged

in *layers*. In this part of the book, we'll focus on networks with three layers, like the one above: an *input layer*, a *hidden layer*, and an *output layer*.

The inputs of the second perceptron come from the first perceptron, except for one: the bias. When we joined two perceptrons together to create the network, both perceptrons kept their bias node. So now we have a bias node in the input layer, and another in the hidden layer. The values of those nodes are fixed at 1. (If you don't remember why 1, then review Bye Bye, Bias, on page 60.)

As I mentioned earlier, the functions in between layers are called the *activation functions* of the network. In the previous diagram, both activation functions are sigmoids—but that isn't necessarily the case, as we'll find out soon.

With that, we know what a three-layers neural network looks like in general. Now let's decide the values of `n`, `d` and `k` for our specific network.

How Many Nodes?

Let's see how many nodes we need for our network's input, hidden, and output layers.

Let's start with the number of input nodes. Just like a perceptron, our network has an input node for each input variable in the data, plus the bias. We're looking to classify MNIST's 784-pixels images, so that makes 785 input nodes. Our input matrix will have one row per image, and 785 columns—the same as the perceptron.

The number of output nodes is also the same as the perceptron's. There are 10 classes in MNIST, so we need 10 output nodes. The output matrix will have one row per image, and 10 columns.

What about the number of hidden nodes? That one is for us to decide, and in a few chapters we'll see how to take a smart decision. For now, let's go with a simple rule of thumb: the number of hidden nodes should be somewhere between the number of input and output nodes. So let's set the number of hidden nodes at 200, that becomes 201 once we add the bias. We're processing all the examples in the training set at once, so the hidden layer will also be a matrix: it will have one row per image, and 201 columns.

One note about the number of hidden nodes: we arbitrarily decided to have 200 hidden nodes, plus one for the bias, for a grand total of 201. Since that initial number is arbitrary, we could just as well start with 199 hidden nodes, and end up with a round 200 after adding the bias. The network is likely to perform pretty much the same with either 200 or 201 hidden nodes—so why

did I opt for an awkward number like 201? The only reason for that choice is that 201 reminds us that one of the hidden nodes is special: it has a fixed value of 1, to take care of the bias.

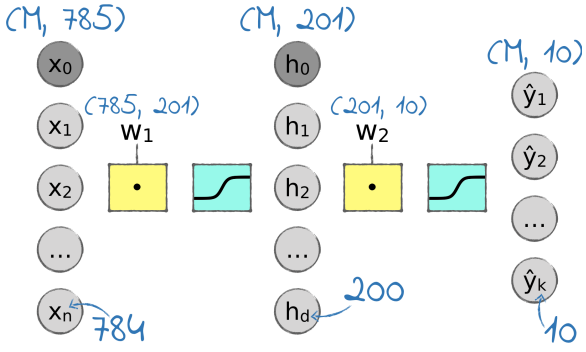
Finally, let's talk about the weights. We built our network by chaining two perceptron, each with its own matrix of weights. As a result, the neural network has two matrices of weights: one between the input and the hidden layer, and one between the hidden and the output layer. We can get their dimensions with a simple general rule: *each matrix of weights in a neural network has as many rows as its input elements, and as many columns as its output elements*. In other words, `w1` is `(n, d)`, and `w2` is `(d, k)`. In case you want to check, that rule also applies to the perceptron: the matrix of weights of our perceptron had one row per input variable, and one column per class.

If you find all these matrix dimensions confusing, check them for yourself. Here are the operations in the network:

```
H = sigmoid(X · W1)
Ŷ = sigmoid(H · W2)
```

Remember the rules of matrix multiplication from Multiplying Matrices, on page 51, and also remember that the output of the sigmoid has the same dimensions as its inputs.

Let's sketch the number of nodes and the matrix dimensions on the network diagram. I will use the letter `m` to indicate the number of inputs, which can vary. It's 60,000 in the MNIST training set, but it can be as little as 1 during classification, if we classify a single image.



The plan is looking nice. You're probably itching to turn it to code—but first, we have one last change to make.

Enter the Softmax

Check out the activation functions in our neural network. So far, we took it for granted that both of those functions are sigmoids. However, most neural networks replace the last sigmoid, the one right before the output layer, with another function called the *softmax*.

Let me show you what the softmax looks like, and then we'll see why it's useful. Like the sigmoid, the softmax takes an array of numbers, that in this case are called the *logits*, and returns an array with the same size as the input. Here is the formula of the softmax, in case you want to understand the math behind it:

$$softmax(l_i) = \frac{e^{l_i}}{\sum e^{l}}$$

You can read this formula as: take the exponential of each logit and divide it by the summed exponentials of all the logits.

You don't need to grok the formula of the softmax, as long as you understand what happens when we use the softmax instead of the sigmoid. Think about the output of the sigmoid in our MNIST classifier, back in Decoding the Classifier's Answers, on page 89. You might remember that for each image in the input, the sigmoid gives us ten numbers between 0 and 1. Those numbers tell us how confident the perceptron is about each classification. For example, if the fourth number is 0.9, that means "this image is probably a 3." If it's close to 0.1, that means "this image is unlikely to be a 3." Amongst those ten results, we pick the one with the highest confidence.

Like the sigmoid, the softmax returns an array where each element is between 0 and 1. However, the softmax has an additional property: the sum of its outputs is always 1. In mathspeak you would say that the softmax *normalizes* that sum to a value of 1.

That's a nice property, because if the numbers add up to 1, then we can interpret them as probabilities. To give you a concrete example, imagine that we're running a three-classes classifier, and the weighted sum after the hidden layer returns these logits:

logit 1	logit 2	logit 3
1.6	3.1	0.5

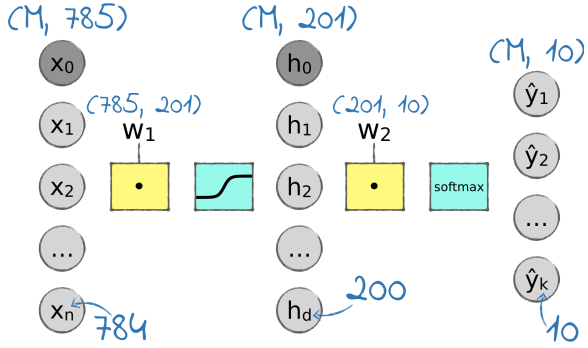
Judging from the numbers above, it seems likely that the data belongs to the second class—but it's hard to gauge *how* likely. Now see what happens if we pass the logits through a softmax:

softmax 1	softmax 2	softmax 3
0.17198205	0.77077009	0.05724785

What's the chance that the data belongs to the second class? Take a glance at the numbers above, and you'll know that the answer is 77%. The first class is way less likely, hovering around 17%, while the third is a measly 6%. Add them together, and you get the expected 100%. Thanks to the softmax, we converted vague numbers to human-friendly probabilities. That's a good reason to use softmax as the last activation function in our neural network.

Here's the Plan

Now that you know about the softmax, let's replace the second sigmoid with a softmax and complete the design of our neural network:



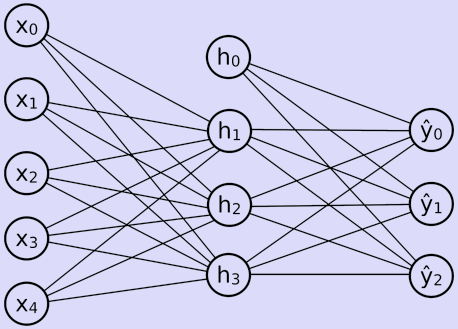
The input layer has 785 columns (one per pixel, plus the bias), and a sigmoid activation function. The hidden layer has 201 columns including the bias, and a softmax activation function. The output layer has 10 columns, matching the 10 classes in MNIST. Finally, the matrices of weights have the right dimensions to make all the matrix multiplications add up. That's all we need to know to get started.

Speaking of code, you must be eager to get down to it. But first, let's recap what we learned in this chapter.

Another Way to Draw Networks

There is no standardized notation for neural networks. This book uses its own simple notation, like the one in Here's the Plan, on page 113. Images of neural networks from

other sources are more likely to use a notation that you might call "blobs and lines". It comes in different flavors, but it generally looks something like this:



The blobs and lines notation does have a few advantages. For example, all those lines give you a good place to jot down the values of individual weights. If you don't need to do that, however, the lines end up cluttering the diagram and making it harder to sketch.

This book uses a more streamlined notation that works well when you sketch neural networks on paper. However, be aware of the blobs and lines notation. It's useful in some edge cases, and it's by far the most common in the wild.

What You Just Learned

In this chapter we designed an *artificial neural network*—or just a *neural network*, for short—by joining two perceptrons. Because the output of the first perceptron is also the input of the second, the resulting neural network has three *layers*: input, hidden, and output.

Each layer is made up of a bunch of *nodes*. In between each pair of consecutive layers, there are two operations: a weighted sum of the first layer, and an *activation function*. In our design, we used a sigmoid as the activation function for the input layer, and a *softmax* for the hidden layer. One last detail: the input and hidden layer also have one bias node each, with a fixed value of 1.

Armed with that knowledge, we designed a network to recognize MNIST characters. In the next chapter we'll turn that design into running code.

Hands On: Network Adventures

We didn't write any code in this chapter, so why don't you try writing it yourself? Here's a task for you: set up a timer—maybe thirty minutes, or one hour, depending on your Python coding experience. Then take the perceptron-

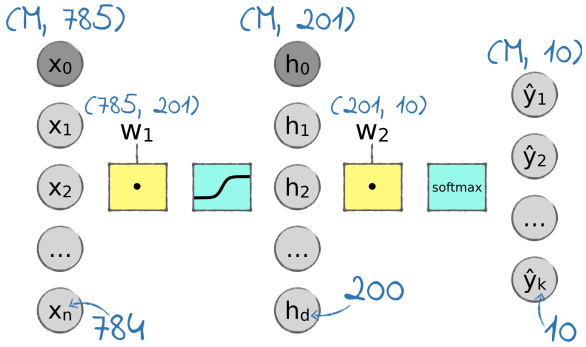
based code of the MNIST reader from Part I of this book, and try to convert it to the neural network that we designed in this chapter. Take as many shortcuts as you need. For example, feel free to ignore the softmax, and just use a second sigmoid instead. Stop when the timer rings.

Just to be clear, nobody expects that you'll complete the job in one hour. In fact, you might have no code at all when the timer rings. That's OK! The point of this exercise is not to code a neural network. The point is to start thinking about the problem, and also to give yourself a review of the perceptron's code. Once the timer rings, turn the page, and let's code that neural network together.

CHAPTER 10

Building the Network

In the previous chapter, we sketched out a neural network to classify MNIST images:



Keep this sketch handy, because we're about to convert it to code. That job will take two chapters. In this one, we'll write the neural network's classification functions—in other words, all the code in the network, except for the training code. In the next chapter, we'll complete the network and unleash it on an unsuspecting MNIST dataset.

Let's get started by writing the neural network's core function.

Coding Forward Propagation

The forward() function was right at the core of our perceptron. It implemented the operation that we called "forward propagation"—running data through the system from the input to the output layer. Let's convert that function for the neural network.

The neural network's forward() is slightly more complicated than the perceptron's. In fact, this is where the name "forward propagation" really comes into

its own: passing an MNIST image through a neural network is like propagating data "forward" through the network's layers, from input to hidden to output.

The first step of forward propagation is exactly the same as a regular perceptron's:

```
h = sigmoid(np.matmul(prepend_bias(X), w1))
```

First, we added a bias column to the inputs; second, we computed the weighted sum of the inputs using the first matrix of weights, `w1`; and third, we passed the result through the `sigmoid()` function. Look at the sketch that opens this chapter, and you'll see that we just calculated the hidden layer.

Now we can repeat that process and calculate the output layer:

```
y_hat = softmax(np.matmul(prepend_bias(h), w2))
```

We used a different matrix of weights, and we passed the weighted sum through `softmax()` instead of `sigmoid()`. (We didn't write `softmax()` yet, but we will soon.) Apart from those differences, the code that calculates the output layer is the same as the one that calculates the hidden layer: add the bias, compute the weighted sum, and pass it through the activation function.

Let's wrap those two lines of code in a function, and let's also borrow the `prepend_bias()` function from our old perceptron code:

```
def forward(X, w1, w2):
    h = sigmoid(np.matmul(prepend_bias(X), w1))
    y_hat = softmax(np.matmul(prepend_bias(h), w2))
    return y_hat
```

There's one last thing to note in the code above. Back when we implemented the perceptron, the `prepend_bias()` function was part of the code that loaded and prepared the data, right in the `mnist.py` library. In the neural network, we need to add the bias twice: once in the input layer, and once in the hidden layer. Because of that, we should move `prepend_bias()` from `mnist.py` to the main network code, so that we can call it multiple times. Here's that function again:

```
def prepend_bias(X):
    return np.insert(X, 0, 1, axis=1)
```

With that, the core forward propagation is done. Now let's fill in the blanks, by writing the `softmax()` function.

Writing the Softmax Function

To complete forward propagation, we need to write the softmax() activation function. As it turns out, we can implement softmax() in two lines of code—but those two lines require some attention.

Here is the mathematical formula of the softmax, straight from Enter the Softmax, on page 112:

$$softmax(l_i) = \frac{e^{l_i}}{\sum e^{l}}$$

And here is that formula converted to code:

```
def softmax(logits):
    exponentials = np.exp(logits)
    return exponentials / np.sum(exponentials, axis=1).reshape(-1, 1)
```

There you have it—more perplexing NumPy code, in case you were missing it. Let's break it down.

Remember, "logits" is the fancy name for the inputs of the softmax. In our case, those inputs are going to be arranged in a matrix that has one line for each MNIST image, and one column per class. For example, if we're training the system with 60,000 images, then softmax() gets a (60000, 10) matrix, and it must return a matrix of the same shape.

The first line in softmax() calculates the exponentials of the logits. The second line divides each exponential by the sum of all the exponentials on the same row. The parameter axis=1 means: "calculate the sum by row, not over the entire matrix." Before doing the division, we must reshape the sums into a one-column matrix—otherwise, NumPy complains that it cannot divide a matrix by a one-dimensional array.

When dealing with NumPy code, the Python interactive interpreter is your friend. Let's fire it up and try out the softmax() function:

```
⇛ output = np.array([[0.3, 0.8, 0.2],
⇛                    [0.1, 0.9, 0.1]])
⇛ softmax(output)
❮ array([[0.28140804, 0.46396343, 0.25462853],
         [0.23665609, 0.52668782, 0.23665609]])
```

See? We still have one row per image and one column per class, but the softmax normalized the logits in each row so that they add up to 1.

That's all about softmax(). However, using a softmax as the last activation function in our network has a subtle consequence: it impacts the loss() function. Let's see why, and how to fix the problem.

Numerical Stability

While playing with the code from this and the next few chapters, you might get an "overflow encountered" warning from NumPy. Those warnings give us the chance to talk about an interesting wrinkle of writing machine learning code.

See what happens if we apply softmax() to logits that have very different values:

```
⇛ softmax(np.array([[1, 20]]))
❮ array([[  5.60279641e-09,   9.99999994e-01]])
```

softmax() exasperated the logits' differences. It returned a tiny value, and a value that's very close to 1. Let's try two logits that diverge further:

```
⇛ softmax(np.array([[1, 1000]]))
❮ __main__:2: RuntimeWarning: overflow encountered in exp
  __main__:3: RuntimeWarning: invalid value encountered in true_divide
  array([[  0.,  nan]])
```

Here is what happened. The first element in the output is so small that it *underflows* Python's math libraries, coming back at 0. The second element involves the exponential of 2000—a huge number that *overflows* Python's math libraries, returning a value of inf (for "infinite"). The softmax() code then tried to divide two infs, resulting in nan (for "not a number"). The first time an overflow occurs, NumPy generates a warning. We can ignore that warning, but we risk flooding our network with nans and zeros.

Underflows and overflows are common when a function includes exponentials or logarithms, like softmax() and sigmoid() do. Those functions are *numerically unstable*: they tend to fail with very large or small input values.

To fix this problem, we could write numerically stable implementations of softmax() and sigmoid(). Those implementations would return the same results as the current ones, but they'd use different formulae to avoid generating those extreme intermediate values. Numerically stable variations are typically more complicated than the plain vanilla functions. For example, a quick Google search reveals this formula for a numerically stable softmax:

$$softmax(l)_i = \frac{e^{l_i + max(e^l)}}{\sum e^{l + max(e^l)}}$$

All that being said, in this book we're going to stick with the plain vanilla versions of our functions, for one reason: numerical instability becomes less of a problem once we stop writing everything from scratch, and start using machine learning libraries. Those libraries already include numerically stable implementations of functions such as softmax() and sigmoid(). Besides, none of the examples in this book generates numbers that cause numerical instability.

If you modify the examples, however, you might stumble upon one of those overflow warnings. Feel free to ignore it: your network will usually work OK even with a few

nans floating around. If you suspect that the network's performance is degrading because of those invalid numbers, then consider writing your own numerically stable implementation of whatever function is failing on you.

Now we have the forward() function, and both activation functions—the sigmoid and the softmax. To complete the classifier, however, we still need to take care of the higher-level classification and testing function: classify() and report(). Let's finish the job.

Writing the Classification Functions

Back in Decoding the Classifier's Answers, on page 89, we wrote a classify() method that predicted the value of an unlabeled image. Here's that method again, modified for the neural network:

```
def classify(X, w1, w2):
    y_hat = forward(X, w1, w2)
    labels = np.argmax(y_hat, axis=1)
    return labels.reshape(-1, 1)
```

The neural network's classify() is the same as the perceptron's, with the exception that it takes two matrices of weights instead of one. Just like the output of a perceptron, the output of the neural network has one row for each image in X, and one column per class—10 in the case of MNIST.

Our perceptron-based MNIST classifier also comes with a report() function that prints the current loss(), and the percentage of correct classifications over the test set. This function can be called either during the classification phase, or during the training phase, to check how well the system is learning. Here's the report() function, updated for the neural network:

```
def report(iteration, X_train, Y_train, X_test, Y_test, w1, w2):
    y_hat = forward(X_train, w1, w2)
    training_loss = loss(Y_train, y_hat)
    classifications = classify(X_test, w1, w2)
    accuracy = np.average(classifications == Y_test) * 100.0
    print("Iteration: %5d, Loss: %.6f, Accuracy: %.2f%%" %
          (iteration, training_loss, accuracy))
```

With that, we wrote all the code for the classification phase of the neural network. However, since we just mentioned the loss() function, it's worth spending a few more words on it.

Cross Entropy

So far, we used the *log loss* formula for our binary classifiers. We even used the log loss when we bundled ten binary classifiers in a multi-class classifier (in Chapter 7, The Final Challenge, on page 85). In that case, we added together the losses of the ten classifiers to get a total loss.

While the log loss served us well so far, it's time to switch to a simpler formula—one that's specific to multi-class classifiers. It's called the *cross entropy loss*, and it looks like this:

$$L = -\frac{1}{m}\sum y \cdot \log(\hat{y})$$

Here's the cross entropy loss in code form:

```
def loss(Y, y_hat):
    return -np.sum(Y * np.log(y_hat)) / Y.shape[0]
```

If you're curious, you can read how the cross entropy loss works in What the Cross Entropy Loss Means, on page 123. However, you don't need to understand how it works, as long as you remember what it does: like other loss formulae, it measures the distance between the classifier's predictions and the labels. The lower the loss, the better the classifier.

Besides its cool name, there is a pragmatic reason to use the cross entropy loss in our neural network: it's a perfect match for the softmax. More specifically, a softmax followed by a cross entropy loss makes it easier to code gradient descent. But I'm getting ahead of myself here—that's a topic for the next chapter. For now, just know that the softmax and the cross entropy loss jive well together. For the rest of this book, the softmax will be the last activation function in our neural networks, and the cross entropy will be our loss formula.

Finally, let me clear one potential source of confusion. If you look at the code, you might wonder why we bother with the loss in the first place. In fact, we don't even seem to ever use the `loss()` function, apart from printing its value on the screen.

Indeed, we don't care about the loss as much as the *gradient* of the loss, that we're going to use later during gradient descent. While the `loss()` function is not really necessary, however, it's still nice to have. We can look at that number to gauge how well the classifier is doing, both during training and during classification. That's the reason why we bothered to code `loss()` and call it from the `report()` function.

What the Cross Entropy Loss Means

I won't delve into the math of the cross entropy loss, but here is an intuitive explanation of what it means. Feel free to skip it if you wish.

The cross entropy is the average over the m examples of this product:

$$-y \cdot \log(\hat{y})$$

In the product above, y is a one hot encoded label, and $\hat{y}$ is the label's inferred value. Remember that a loss measures how wrong the inferred value is. If $\hat{y}$ points at the same label as y, then the loss is 0. The farther $\hat{y}$ is from y, the bigger the loss.

Now consider that, because of one hot encoding, y has one element per label—but only one of those elements contains a 1. The other elements are zeros, so they don't really matter when calculating the loss. Their contribution to the product $-y \cdot \log(\hat{y})$ is always 0, whatever the value of $\hat{y}$.

For the element where y is 1, however, the matching element $\hat{y}$ becomes important. If the classifier set it at 1 (meaning: "I'm pretty sure that this is the right label"), then we're golden: the loss becomes $-1 \cdot \log(1)$, which is 0. Otherwise, $-1 \cdot \log(\hat{y})$ becomes a positive number. Average those numbers over all the examples, and there you have it—our total loss.

From now on, the cross entropy loss replaces the "old" log loss that I introduced in Smoothing It Out, on page 69. However, the two losses are closely related. If you like to play with math, you can even prove that the log loss is nothing but a binary version of the cross entropy loss: if you have only two labels, then the log loss and the cross entropy loss become identical.

With the loss() function, our neural network's classification code is done. Let's wrap it up.

A Quick Code Review

Here is all the code we have written so far in this chapter:

10_building/forward_propagation.py

```
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def softmax(logits):
    exponentials = np.exp(logits)
    return exponentials / np.sum(exponentials, axis=1).reshape(-1, 1)

def loss(Y, y_hat):
    return -np.sum(Y * np.log(y_hat)) / Y.shape[0]
```

```
def prepend_bias(X):
    return np.insert(X, 0, 1, axis=1)

def forward(X, w1, w2):
    h = sigmoid(np.matmul(prepend_bias(X), w1))
    y_hat = softmax(np.matmul(prepend_bias(h), w2))
    return y_hat

def classify(X, w1, w2):
    y_hat = forward(X, w1, w2)
    labels = np.argmax(y_hat, axis=1)
    return labels.reshape(-1, 1)

def report(iteration, X_train, Y_train, X_test, Y_test, w1, w2):
    y_hat = forward(X_train, w1, w2)
    training_loss = loss(Y_train, y_hat)
    classifications = classify(X_test, w1, w2)
    accuracy = np.average(classifications == Y_test) * 100.0
    print("Iteration: %5d, Loss: %.6f, Accuracy: %.2f%%" %
          (iteration, training_loss, accuracy))
```

With that, half of our neural network's logic is done—the half that takes care of the classification phase.

All this code, however, won't be useful until we've also written the code that trains the network. In the next chapter we'll do that, and we'll test drive the entire thing.

Hands On: Time Travel Testing

We wrote our neural network's classification code, but not its training code. For that, you'll have to wait until the next chapter. However, at this point you might be itching to see the network running. How can we run the classification code that needs pre-calculated weights if we don't have the training code that finds the values of those weights?

We can solve this problem thanks to time travel. Book authors have mastered that elusive skill—and guess what: I just returned from the future, where I have access to the complete neural network code. After training the complete network for a few iterations, I serialized the two matrices of weights to a JSON file named weights.json, that you can find amongst this chapter's source code.

Go forth, give it a try. Add this code to forward_propagation.py and run it:

```
if __name__ == "__main__":
    import json
    with open('weights.json') as f:
        weights = json.load(f)
    w1, w2 = (np.array(weights[0]), np.array(weights[1]))
```

```
import mnist
report(0, mnist.X_train, mnist.Y_train, mnist.X_test, mnist.Y_test, w1, w2)
```

This code uses Python's json.load() function to deserialize w1 and w2 from the weights.json file. Then it runs a classification on the MNIST test set. If you try it yourself, you should get a bit more than 43% correct matches.

That 43% is nothing to write home about—but remember, these are the weights that I got after just a handful of training iterations. To see how good the neural network gets after some serious training, you'll have to wait for the next chapter. How's that for a cliffhanger?

CHAPTER 11

Training the Network

It's time to work on the last piece of our neural network: the training code.

In the early years of neural networks, training was a tough nut to crack. Researchers even questioned whether they could be trained at all. The answer came in 1970, when a Finnish researcher trained a network with an algorithm called *backpropagation*—or "backprop" for friends.

This chapter is all about backpropagation: how it works and how to make it run smoothly. If you rely on modern ML libraries, then you're unlikely to ever implement backprop yourself. Still, you should get an intuitive sense of this algorithm so that you're well equipped to deal with its subtle consequences.

In the next few pages, I'll introduce backpropagation, and we'll implement it for our neural network. You'll also learn to initialize the network's weights so that they jive well with backprop. At the end of this chapter, we'll put the network through its maiden run. Will it beat the perceptron's accuracy? If so, by how much?

A Quick Head's Up

A few backprop tutorials open with a bold declaration: "backpropagation is easy!" I beg to disagree. Backpropagation is pretty darn hard. It only becomes easy in retrospect, *after* you know how it works.

To work over that initial hump, this chapter focuses on intuition. We'll grok the general workings of backprop, but we'll skip a few details and long-winded calculations.

I strived to keep these explanations as simple as I could, but not simpler. Together with Chapter 4, Hyperspace!, on page 47, this is the most math-heavy chapter in the book. Don't feel frustrated if you don't understand all of it on your first read through. Most people take time to wrap their minds around backpropagation. I know I did!

Backpropagation

This section explains backpropagation from the ground up. We'll go through a series of examples: a basic one that shows the core idea of backpropagation; a more complex one that involves a tiny neural network; and finally, we'll apply backprop to our own network.

But first, let's explore why we use backpropagation in the first place.

Why We Use Backpropagation

We already said that people use backpropagation to train neural networks, but we didn't explain how. Let's dig a bit deeper.

To begin with, here's a piece of good news: in a sense, you already know how to train a neural network. You train it with gradient descent, like you train a perceptron. At each iteration, you calculate the gradient of the loss, then descend that gradient to minimize the loss. (If you need a refresher, review Chapter 3, Walking the Gradient, on page 33.)

Now for the less-than-good news: descending the gradient is the easy part of the job. The tough part is calculating that gradient in the first place.

In the case of the perceptron, we knew how to get the gradient: we calculated the derivatives of the loss with respect to the weights. (Well, we actually looked up those derivatives in a textbook. Still counts.) In the case of a neural network, however, coming up with those derivatives can be hard. For example, the code that computes the loss in our three-layers network looks something like this:

```
h = sigmoid(matmul(X, w1))
y_hat = softmax(matmul(h, w2))
L = cross_entropy_loss(Y, y_hat)
```

Imagine writing the mathematical equivalent of the previous code. We'd have to expand the sigmoid, the softmax, and the cross entropy loss. The resulting formula would be large and unwieldy. Then we'd have to take that formula's derivatives with respect to w_1 and w_2. That'd be work, even for someone steeped in calculus.

We could still sweat our way to the gradient of our neural network—but the typical modern network would be much more challenging. Real-world neural networks can be mindbogglingly complicated, with dozens of intricately connected layers and weight matrices. It would be very hard to write down the loss for one of those large networks—let alone calculate its derivatives.

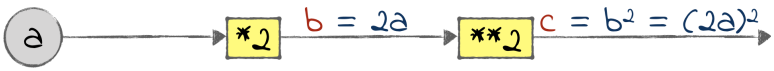
We're facing a dilemma: on one hand, we want to calculate the gradients of the loss for any neural network; on the other, taking those derivatives is unfeasible, except for the simplest and less powerful networks.

That's where backpropagation enters the picture. Backprop is the way out of that dilemma—an algorithm that calculates the gradients of the loss in any neural network. Once we know those gradients, we can descend them with good old GD.

Let's see how backprop works its magic.

The Chain Rule

Backpropagation is an application of the *chain rule*, one of the fundamental rules of calculus. Let's see how the chain rule works with a visual example. Look at this simple network-like structure:



The one above isn't a neural network, because it doesn't have weights. Let's borrow a term from computer science, and call it a *data flow diagram*. This graph has an input `a`, followed by two operations: "multiply by two" and "square." The output of the multiplication is called `b`, and the output of the entire graph is called `c`.

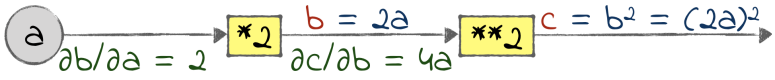
Now let's say that we want to calculate `∂c/∂a`, the gradient of `c` with respect to `a`. Intuitively, that gradient represents the impact of `a` on `c`: whenever `a` changes, `c` also changes, and the gradient measures how much. (If this way of thinking about the gradient sounds perplexing, then maybe review Gradient Descent, on page 35.)

For such a small graph, we could calculate `∂c/∂a` in a single shot, by taking the derivative of `c` with respect to `a`. As we mentioned earlier, however, that derivation would become impractical for very large graphs. Instead, we can calculate the gradient using the chain rule, that works for graphs of any size.

Here is how the chain rule works. To calculate `∂c/∂a`:

1. Walk the graph back from `c` to `a`.
2. For each operation along the way, calculate its *local gradient*—the derivative of the operation's output with respect to its input.
3. Multiply all the local gradients together.

Let's see how that process works in practice. In our case, the path back from c to a involves two operations: a square and a multiplication by 2. Let's jot down the local gradients of those two operations:



How do I know that ∂b/∂a is 2, and ∂c/∂b is 4a? Well, even though we're using the chain rule, we must still compute the local gradients in the old-fashioned way, taking derivatives by hand. However, don't fret if you don't know how to take derivatives—there are libraries that do that. Just understand how the process works, and you're golden.

Now that we have the local gradients, we can multiply them to get ∂c/∂a:

$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b} \cdot \frac{\partial b}{\partial a} = 4a \cdot 2 = 8a$$

So, there's our answer, courtesy of the chain rule: the gradient of c with respect to a is 8a. In other words, if a changes a little, then c changes by 8 times the current value of a.

To recap the chain rule: to calculate the gradient of any node y with respect to any other node x, we multiply the local gradient of all the nodes on the way back from y to x. Thanks to the chain rule, we can calculate a complicated gradient as a multiplication of many simple gradients.

Now let's apply the chain rule to a neural network.

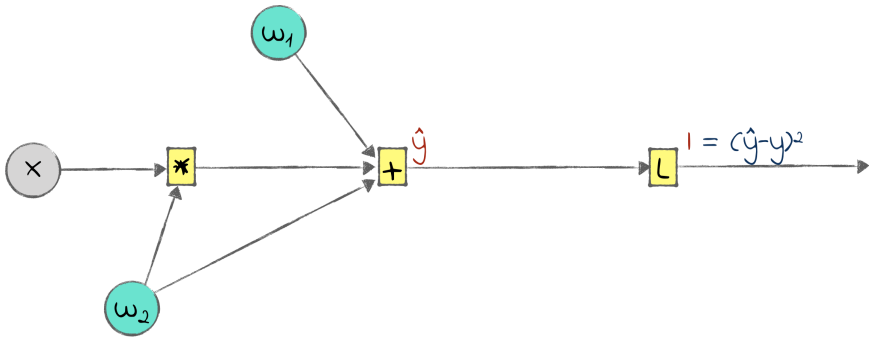
Math Deep Dive: The Chain Rule

If you watched Khan Academy's screencasts on calculus, then you might already have seen the videos on the chain rule.[a] As usual, this material goes deeper than you need to read this chapter—but if you like math, it's definitely worth a watch.

a. www.khanacademy.org/math/differential-calculus/dc-chain

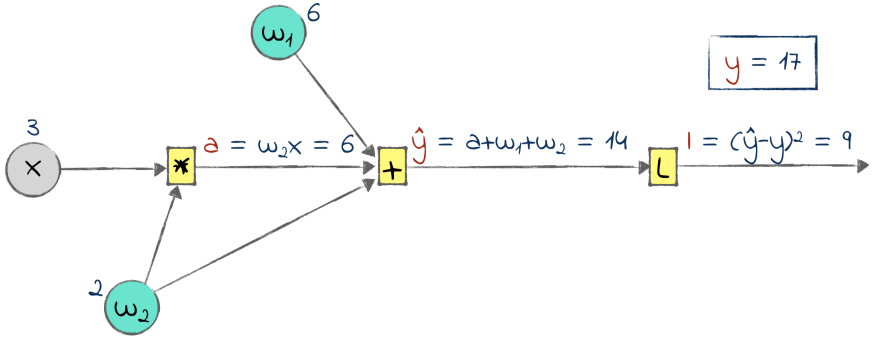
Understanding Backpropagation

The name "backpropagation" is a shorthand for "calculate the gradients of a neural network's loss with respect to the weights using the chain rule." As an example, check out this second data flow diagram:



This graph is not going to win any machine learning competitions. In fact, you might argue that it's not even a neural network. However, it has what it takes to stand for a neural network in this example: an input `x`, an output $\hat{y}$, and a couple of weights. It also has a loss `L`, calculated as the squared error of the difference between $\hat{y}$ and the expected output `y`.

Imagine freezing this network mid-training, right before the next step of gradient descent. Let's say that w_1 and w_2 are currently 6 and 2. Also, let's say that we only have one training example, that has `x = 3` and `y = 12`. From those numbers, we can calculate the other values in the graph:

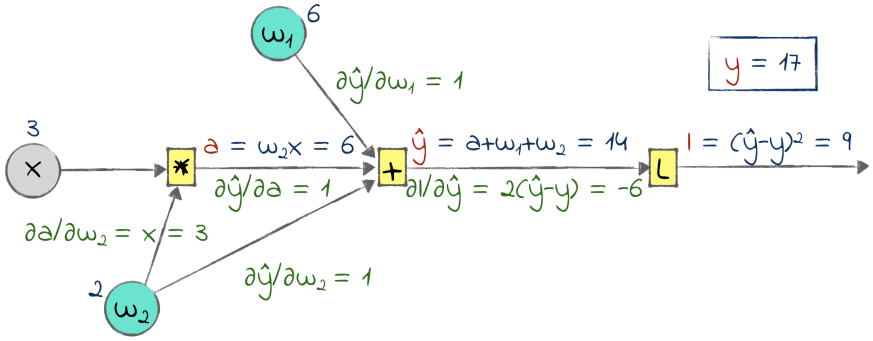


I didn't have a name for the output of the multiplication, so I called it `a`.

To take the next step of GD, we need the gradients of `L` with respect to w_1 and w_2. We can compute those gradients with the chain rule. Remember how it works? $\partial L/\partial w_1$ is the product of the local gradients on the way back from `L` to w_1. The same goes for $\partial L/\partial w_2$. Let's get down to business and calculate those local gradients.

Note that this graph has an added difficulty compared with the one from the previous section: some operations have multiple inputs. If an operation has multiple inputs or outputs, then we have to calculate its local gradient for

each input-output pair. Taking that fact into account, we need five local gradients. Here they are, complete with their numerical values:



Now we can apply the chain rule. First, let's calculate ∂L/∂w₁. If you wanted to be precise, you could read that gradient as "the gradient of L with respect to w₁"—but because that's a mouthful, machine learning practitioners tend to call it "the gradient of w₁", for short.

To get the gradient of w₁, we multiply all the gradients between L and w₁:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1} = -6 \cdot 1 = -6$$

There you have it: with the current weights, the gradient of w₁ is -6. To take the next step of GD, we would multiply this gradient by the learning rate, and subtract the result from w₁.

We can follow a similar process to get ∂L/∂w₂, although in this case we have an additional complication: there are two paths leading from w₂ to L—one that passes by the multiplication, and one that doesn't. Whenever we have multiple paths, we have to sum their gradients:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2} + \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{\partial w_2} = -6 \cdot 1 + -6 \cdot 1 \cdot 3 = -24$$

∂L/∂w₂ is also negative, but larger than ∂L/∂w₁. Once again, we can multiply this gradient by the learning rate, and subtract the result from w₂.

You can see ∂L/∂w₁ and ∂L/∂w₂ as measures of how much each weight is contributing to the loss. Both weights have a negative contribution, meaning that they must grow so that the loss gets smaller. However, w₂ is contributing more than w₁, because it's involved twice in the calculation of ŷ—once in the multiplication, and once in the sum.

In general, if a weight has a small gradient, that means that it doesn't contribute much to the network's error, so it can change just a little bit. Converse-

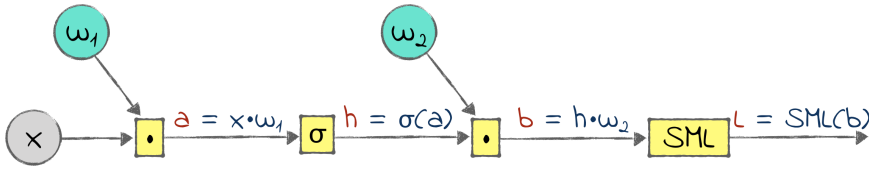
ly, a weight with a large gradient is having a big impact on the network's error, and needs to be changed more decisively. Backprop is a way to calculate how much each weight needs to change.

We just applied backpropagation—an algorithm to calculate the gradients of the weights by multiplying the local gradients of individual operations. It's called *back*propagation because, conceptually, it moves in the opposite direction of forward propagation. Forward propagation moves from the inputs to the outputs, and ultimately calculates the loss; backpropagation moves back from the loss to the weights, accumulating local gradients through the chain rule.

Now that we have a grip on backpropagation, let's apply it to our network.

Applying Backpropagation

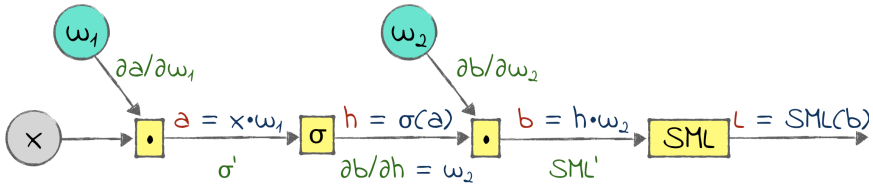
Here is our three-layered network, in the form of a data flow diagram:



All the variables in the preceding graph are matrices, with the exception of the loss `L`. `σ` represents the sigmoid. For reasons that will become clear in a minute, I squashed the softmax and the cross entropy loss into one operation called `SML`, that stands for "softmax and loss." I also gave the names `a` and `b` to the outputs of the matrix multiplications.

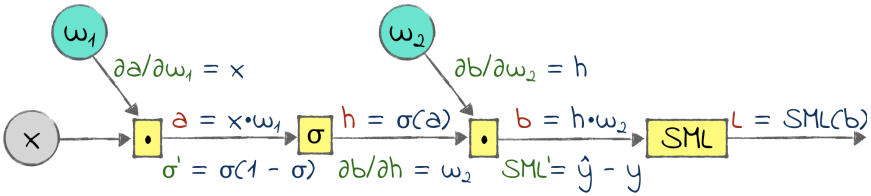
The diagram above represents the same neural network that we designed and built in the previous two chapters. Follow its operations from left to right: `x` and w_1 get multiplied and passed through the sigmoid, producing the hidden layer `h`; then, `h` is multiplied by w_2 and passed through the softmax and the loss function, producing the loss `L`.

Now let's come to the reason we learned backpropagation: we want to calculate the gradients of `L` with respect to w_1 and w_2. To apply the chain rule, we need the local gradients on the paths back from `L` to w_1 and w_2:



The apostrophe is another common way to mean "derivative," so σ' and SML' are the derivatives of the sigmoid and the SML operation, respectively.

Let's roll up our sleeves and calculate those local gradients. Once again, you don't need to be able to take those derivatives on your own. That's what I'm here for! Here are the results:



Back in Cross Entropy, on page 122, I said that the softmax and the cross entropy loss are a perfect match. Now we can see why. Taken separately, those functions have complicated derivatives—but compose them, and their derivative boild down to a simple gradient that's cheap to compute: the network's output minus the expected output.

The derivatives of the matrix multiplications are also short and sweet. If you know how to derive scalar multiplication—well, the derivatives of matrix multiplication are the same.

Finally, I looked up the sigmoid's derivative in a math textbook. It's a peculiar one, because it's expressed in terms of the sigmoid itself.

Now that we have the local gradients, we can apply the chain rule to calculate the gradients of the weights. This time, however, we're not going to do on paper—we're going to write code.

Staying the Course

We're about to write a backpropagation function that calculates the gradients of the weights in our neural network. Here is a word of warning before we begin: this function is going to be short, but complicated.

We will write code that's a direct application of the chain rule. The chain rule is straightforward when you apply it to scalar gradients, but gets complicated with matrices. In a neural network, you have to "make the multiplications work" by swapping and transposing the operands, so that the dimensions of the matrices involved work well together.

That juggling of matrices is actually easier to do than it is to describe. When you write your own backprop code, you can double-check your calculations in an interpreter, and use the matrix dimensions as a guideline to decide

which matrices to transpose, in which order to multiply them, and so on. In written form, however, the reasoning behind those operations takes up more space than we have available in this chapter.

All those things considered, now you have a few options:

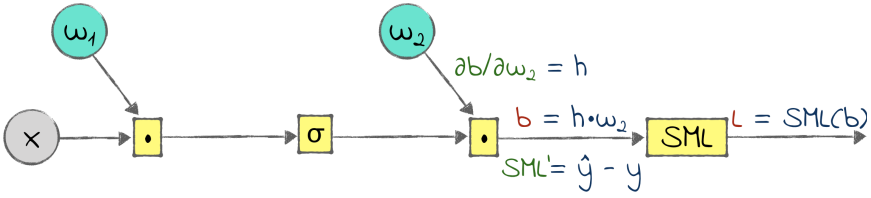
1. You can read through the rest of this section and see how the backprop code comes together, breezing over the twisty details.
2. You can go through the next few pages more carefully, double-checking the matrix operations on your own, either on paper or in a Python interpreter.
3. If you're looking for some hard work, you can even write the backpropagation code yourself, peeking at the next pages for hints if you get stuck. This last option is challenging, but it's the best way to wrap your head around all the details of this code, if you're so inclined.

Whichever option you choose, don't get discouraged if you struggle through the corner cases in the next two pages. Those corner cases aren't the point of this chapter. The point is to understand the idea behind backpropagation, and get an idea of how this code came together.

Preamble done. Let's write the code that calculates the gradient of w_2.

Calculating the Gradient of w_2

Here are the local gradients involved in the calculation of $\partial L/\partial w_2$:



Apply the chain rule with an eye on the diagram above, and you get:

$$\frac{\partial L}{\partial w_2} = \text{SML}' \cdot \frac{\partial b}{\partial w_2} = (\hat{y} - y) \cdot h$$

And here is that formula turned to code:

11_training/backpropagation.py

```
w2_gradient = np.matmul(prepend_bias(h).T, y_hat - Y) / X.shape[0]
```

I had to swap the operands of the multiplication, and transpose one of them—all in the interest of getting a result with the same dimensions as w_2. Remember that we're going to multiply this gradient by the learning rate, and subtract it from w_2, so the two matrices must have the same shape.

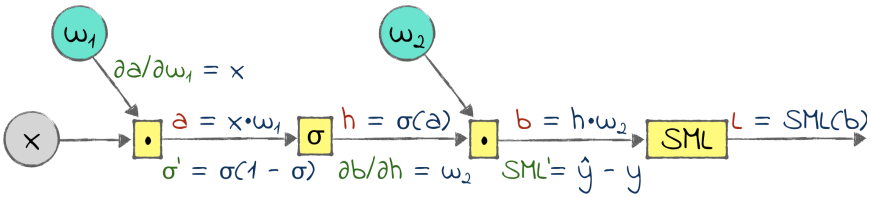
The one-liner above contains two more details that you might find baffling. One is the call to `prepend_bias()`. To make sense of it, remember that in the `forward()` function, we add a bias column to `h`, so we need to do the same during backprop.

The second perplexing detail is the division by `X.shape[0]` at the end. That's the number of rows in `X`—that is, the number of examples in the training set. This division is there because the matrix multiplication gives us the accumulated gradient over all the examples. Because we want the *average* gradient, we have to divide the result of the multiplication by the size of the training set.

That was a tightly packed line of code. On to the gradient of w_1!

Calculating the Gradient of w_1

Let's look at the local gradients that we need to calculate $\partial L/\partial w_1$:



Here comes the chain rule:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2} + \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{\partial w_2} = -6 \cdot 1 + -6 \cdot 1 \cdot 3 = -24$$

That's a long formula. I split it over two lines of code and a helper function:

```
def sigmoid_gradient(sigmoid):
    return np.multiply(sigmoid, (1 - sigmoid))

a_gradient = np.matmul(y_hat - Y, w2[1:].T) * sigmoid_gradient(h)
w1_gradient = np.matmul(prepend_bias(X).T, a_gradient) / X.shape[0]
```

The next to last line calculates $(\hat{y} - y) \cdot w_2 \cdot \sigma$. There are a few subtleties involved in this calculation. To begin with, this time around we're using `h` as is, without a bias column. Fact is, the bias column gets added *after* the calculation of `h`, so its gradient doesn't propagate as far back as the gradient of `w1`. In other words, that column has no effect on $\partial L/\partial w_1$.

Now, here's a twist: if we remove the first column of `h`, then we also have to remove its weights. That's the first *row* of `w2`, because matrix multiplication matches columns by rows. That's what `w2[1:]` means: "`w2`, without the first row".

I have to admit that I got this code wrong the first time. After NumPy complained that my matrix dimensions didn't match, I fixed the code by thinking through the dimensions of all the matrices involved. It was a pretty painful experience.

Moving on with this line of code: the sigmoid_gradient() helper function calculates the sigmoid's gradient from the sigmoid's output. We already know that the sigmoid's output is the hidden layer h, so we can just call sigmoid_gradient(h).

The last line in the code above finishes the job, multiplying the previous intermediate result by X. This multiplication involves the same trickery that we experienced when we calculated the gradient of w_2: we need to swap the operands, transpose one of the matrices, prepend the bias column to X like we do during forward propagation, and average the final gradient over the training examples.

Phew! I told you that this code was going to be tricky, but we're done now. We can put it all together.

Distilling the back() Function

Here is the backpropagation code, inlined and wrapped into a convenient three-lines function:

```
def back(X, Y, y_hat, w2, h):
    w2_gradient = np.matmul(prepend_bias(h).T, (y_hat - Y)) / X.shape[0]
    w1_gradient = np.matmul(prepend_bias(X).T, np.matmul(y_hat - Y, w2[1:].T)
                            * sigmoid_gradient(h)) / X.shape[0]
    return (w1_gradient, w2_gradient)
```

Congratulations! We're done writing the hardest piece of code in this book.

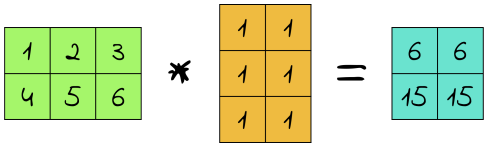
Now that we have the back() function, we're one big step closer to a fully functioning GD algorithm for our neural network. We just need to take care of one last detail: before we start tuning the weights, we have to initialize them. That matter deserves its own section.

Initializing the Weights

Let me wax nostalgic about perceptrons for a moment. Back in Part I of this book, weight initialization was a quick job: we just set all the weights to 0. By contrast, weight initialization in a neural network comes with a hard-to-spot pitfall. Let's describe that pitfall, and see how to walk around it.

Fearful Symmetry

Here is one rule to keep in mind: never initialize all the weights in a neural network with the same value. The reason for that recommendation is subtle, and comes from the matrix multiplications in the network. For example, look at this matrix multiplication here:



You don't need to remember the details of matrix multiplication here, although you can review Multiplying Matrices, on page 51 if you want to. Here is the interesting detail in the example above: even though the numbers in the first matrix are all different, the result has two identical rows, because of the uniformity of the second matrix. In general, matrix multiplication tends to preserve symmetry.

Now imagine that the first and second matrices above are respectively `x` and w_1—the inputs and the first layer's weights of a neural network. Once the multiplication is done, the resulting matrix passes through a sigmoid to become the hidden layer `h`. Now `h` has the same values in each row, meaning that all the hidden nodes of the network have the same value. By initializing all the weights with the same value, we forced our network to behave as if it had only one hidden node.

If w_2 is also initialized uniformly, this symmetry-preserving effect happens on the second layer as well, and even during backpropagation. In the "Hands On" section at the end of this chapter, we'll get a chance to peek into the network and see how uniform weights cause the network to behave like a one-node network. As you might guess, such a network isn't very accurate. After all, there is a reason why we have all those hidden nodes to begin with.

We don't want to choke our network's power, so we shouldn't initialize its weights to 1, or any other constant value such as 0. Instead, we should initialize the weights to random values. ML practitioners have a cool name for this random initialization: they call it "breaking the symmetry."

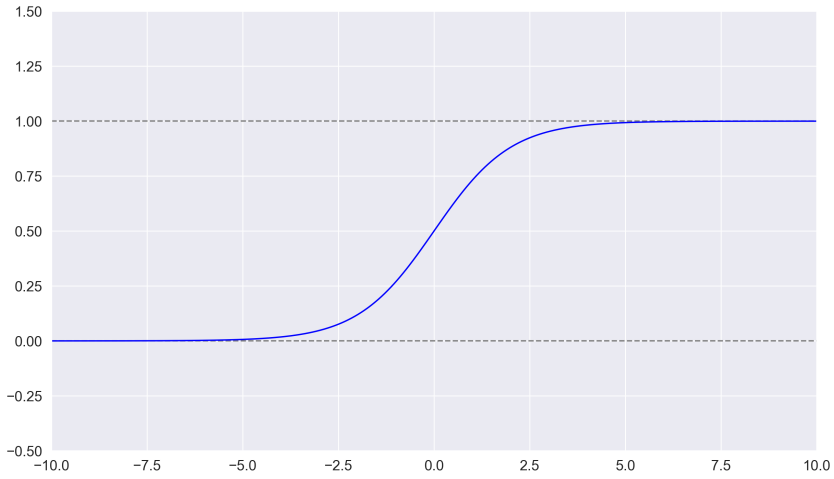
To wrap it up, we should initialize the neural network's weights with random values. The next question is: how large or small should those values be? That innocent question opens up yet another can of worms.

Dead Neurons

We learned that we shouldn't initialize our weights uniformly. However, there is also another rule that we should take to heart: we shouldn't initialize the weights with large values.

There are two reasons for that rule of thumb. One reason is that large weights generate large values inside the network. As we discussed in Numerical Stability, on page 120, large values can cause problems if the network's function are not numerically stable: they can push those functions past the brink, making them overflow.

Even if all the functions in your network are numerically stable, large weights can still cause another subtler problem: they can slow down the network's training, and even halt it completely. To understand why, take a glance at the sigmoid function that sits right at the core of our network:



Consider what happens if one of the weights in w_1 is very large—either positive or negative. In that case, the sigmoid gets a large input. With a large input, the sigmoid becomes a very flat function, with a gradient close to 0. In technical terms, the sigmoid gets *saturated*: it's pushed outside of its ideal range of operation, to a place where its gradient becomes tiny.

During backpropagation, that tiny gradient gets multiplied by all the other gradients in the chain, resulting in a small overall gradient. In turn, that small gradient forces gradient descent to take very tiny steps.

To summarize this dismaying chain of cause and effect: the larger the weight, the flatter the sigmoid; the flatter the sigmoid, the smaller the gradient; the

smaller the gradient, the slower GD; and the slower GD, the less the weight changes. A handful of large weights are enough to slow the entire training to a crawl.

If the numbers in the network get even larger, things go south fast. Imagine that the network has an input with a value of 10, and a weight that's around 1000. See what happens when those weight and input are multiplied and fed to the sigmoid:

```
⇛ import neural_network as nn
⇛ weighted_sum = 1000 * 10
⇛ nn.sigmoid(weighted_sum)
❮ 1.0
```

The sigmoid *saturated*—meaning that it got so close to 1, that it underflowed and just returned 1. Even worse, its gradient underflowed to 0:

```
⇛ nn.sigmoid_gradient(nn.sigmoid(weighted_sum))
❮ 0.0
```

Remember the chain rule? During backpropagation, this gradient of 0 gets multiplied by the other local gradients, causing the entire gradient to become 0. With a gradient of 0, gradient descent has nowhere to go. This weight is never going to change again, no matter how long we train the network. In machine learning lingo, the weight has become a *dead neuron*.

In the first page of this chapter, I mentioned that backpropagation comes with a few subtle consequences. Dead neurons are one of them. A dead neuron is stuck to the same value forever. It never learns, and doesn't contribute to GD. If too many neurons die, the network loses power—and worse, you might not even notice.

So, how do we prevent dead neurons?

Weight Initialization Done Right

Let's wrap up what we said in the previous sections. We should initialize weights with values that are:

1. *random*, to break symmetry;
2. *small*, to speed up training and avoid dead neurons.

It's hard to gauge how small exactly the weights should be. In Part III of this book, we'll look at a few popular formulae to initialize weights. For now, we can use an empirical rule of thumb. We'll make each weight range from 0 to something around this value, where `r` is the number of rows in the weight matrix:

$$w \approx \pm\sqrt{(1/r)}$$

Read it as: the absolute value of each weight shouldn't be much bigger than the square root of the inverse of rows. That idea makes more sense if you think that all weights add something to the network's output—so, as the weight matrix gets bigger, individual weights should become smaller.

Here is a function that initializes the weights with the previous formula:

11_training/neural_network.py

```
def initialize_weights(n_input_variables, n_hidden_nodes, n_classes):
    w1_rows = n_input_variables + 1
    w1 = np.random.randn(w1_rows, n_hidden_nodes) * np.sqrt(1 / w1_rows)

    w2_rows = n_hidden_nodes + 1
    w2 = np.random.randn(w2_rows, n_classes) * np.sqrt(1 / w2_rows)

    return (w1, w2)
```

NumPy's random.randn() function returns a matrix of random numbers taken from what is called the *standard normal distribution*. In practice, that means that the random numbers are either positive or negative, but unlikely to become much bigger than 1 or 2. The code above creates two such matrices of weights, and then scales them by the range suggested by our rule-of-thumb formula.

If you're confused by the dimensions of the two matrices, remember the rule that we mentioned in (#sec.how_many_weights): each matrix of weights in a neural network has as many rows as its input elements, and as many columns as its output elements.

Did you hear that "click" sound? That was the last piece of our neural network's code falling into place. We're finally about to run this thing! But first, let's recap what we learned in this long chapter.

Getting Used to Randomness

If you initialize a neural network with random weights, then you should expect a slightly different loss every time you train it. If you wish, you can avoid that randomess by seeding NumPy's random number generator with np.random.seed(a_known_seed).

During my first ML experiences, I always seeded the random generator. I strived for deterministic code that gave the same result every time. Over time, I learned that a good classifier stays good no matter how you initialize its weights, as long as you use small enough values. Like most ML practitioners, I learned to stop worrying and embrace the randomness.

What You Just Learned

In this chapter, you learned *backpropagation*—an algorithm to calculate the gradients of the weights in a neural network. Those gradients represent the impact of each weight on the overall loss.

We also learned that neural networks don't train well if all the weights have the same value. Instead, we should *break their symmetry* by initializing the weights with random values. Those initial values should also be small to avoid problems like *overflows* and *dead neurons*.

Once the weights have been initialized, each training iteration ping-pongs between two steps:

1. *Forward propagation*: the network calculates each layer from the previous one, from the input layer to the output $\hat{y}$.
2. *Backpropagation*: the network bounces its way back from the output layer to the weights, using the *chain rule* to calculate their gradients. Then it descends those gradients, pushing the loss down, and $\hat{y}$ closer to the expected output y.

Let's look at this process in action.

The Finished Network

We're ready to review and run our neural network. Let's take a look at the entire thing.

The Whole Shebang

This is the complete code of our neural network:

11_training/neural_network.py

```
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def softmax(logits):
    exponentials = np.exp(logits)
    return exponentials / np.sum(exponentials, axis=1).reshape(-1, 1)

def sigmoid_gradient(sigmoid):
    return np.multiply(sigmoid, (1 - sigmoid))

def loss(Y, y_hat):
    return -np.sum(Y * np.log(y_hat)) / Y.shape[0]
```

```
def prepend_bias(X):
    return np.insert(X, 0, 1, axis=1)

def forward(X, w1, w2):
    h = sigmoid(np.matmul(prepend_bias(X), w1))
    y_hat = softmax(np.matmul(prepend_bias(h), w2))
    return (y_hat, h)

def back(X, Y, y_hat, w2, h):
    w2_gradient = np.matmul(prepend_bias(h).T, (y_hat - Y)) / X.shape[0]
    w1_gradient = np.matmul(prepend_bias(X).T, np.matmul(y_hat - Y, w2[1:].T)
                            * sigmoid_gradient(h)) / X.shape[0]
    return (w1_gradient, w2_gradient)

def classify(X, w1, w2):
    y_hat, _ = forward(X, w1, w2)
    labels = np.argmax(y_hat, axis=1)
    return labels.reshape(-1, 1)

def initialize_weights(n_input_variables, n_hidden_nodes, n_classes):
    w1_rows = n_input_variables + 1
    w1 = np.random.randn(w1_rows, n_hidden_nodes) * np.sqrt(1 / w1_rows)

    w2_rows = n_hidden_nodes + 1
    w2 = np.random.randn(w2_rows, n_classes) * np.sqrt(1 / w2_rows)

    return (w1, w2)

def report(iteration, X_train, Y_train, X_test, Y_test, w1, w2):
    y_hat, _ = forward(X_train, w1, w2)
    training_loss = loss(Y_train, y_hat)
    classifications = classify(X_test, w1, w2)
    accuracy = np.average(classifications == Y_test) * 100.0
    print("Iteration: %5d, Loss: %.8f, Accuracy: %.2f%%" %
          (iteration, training_loss, accuracy))

def train(X_train, Y_train, X_test, Y_test, n_hidden_nodes, iterations, lr):
    n_input_variables = X_train.shape[1]
    n_classes = Y_train.shape[1]
    w1, w2 = initialize_weights(n_input_variables, n_hidden_nodes, n_classes)
    for i in range(iterations):
        y_hat, h = forward(X_train, w1, w2)
        w1_gradient, w2_gradient = back(X_train, Y_train, y_hat, w2, h)
        w1 = w1 - (w1_gradient * lr)
        w2 = w2 - (w2_gradient * lr)
        report(i, X_train, Y_train, X_test, Y_test, w1, w2)
    return (w1, w2)

if __name__ == "__main__":
    import mnist
```

```
w1, w2 = train(mnist.X_train, mnist.Y_train,
               mnist.X_test, mnist.Y_test,
               n_hidden_nodes=200, iterations=10000, lr=0.01)
```

To write the very last line above, I had to set values for the hyperparameters—in particular, the learning rate `lr`. I tried a few different values for it, and used the one that resulted in the lowest loss. Near the end of Part II, we'll look at a more systematic way to pick hyperparameter values.

Let's run this network, at long last.

The Moment You've Been Waiting For

Here are the results of running the network:

```
Iteration:     0, Loss: 2.38746031, Accuracy: 13.61%
Iteration:     1, Loss: 2.34527197, Accuracy: 15.00%
…
Iteration:  9999, Loss: 0.14668400, Accuracy: 93.25%
```

If you look back at Chapter 7, The Final Challenge, on page 85, you'll see that the perceptron couldn't reach 92% accuracy, no matter how long you trained it. The neural network has to do more calculations than a perceptron, so it takes a bit longer to pass 90% accuracy—but after that, it just keeps going. After a few hours of number crunching, the network has reached over 93% accuracy in character recognition on MNIST. Pretty good, although still far from our lofty goal of 99% accuracy.

In case you think that my enthusiasm for that 1% improvement is unwarranted, look at it from the opposite point of view: the perceptron makes a classification mistake over 8% of the times, while the network gets it wrong less than 7% of the times. That's a big improvement for a production system. Over the next few chapters, we'll push that accuracy much higher.

We deserve a little rest after all this detail-oriented work. In the next chapter, we'll consider the big picture. In particular, we'll ask ourselves a question that we haven't really answered so far: how the heck do neural networks work?

Hands On: Starting Off Wrong

In Fearful Symmetry, on page 138, we learned that we shouldn't initialize all the neural network's weights to the same value. See for youself what happens if we ignore that advice.

You can use NumPy's `zeros()` function to initialize all the weights to 0. For example, here is how you get a matrix of zeros with 2 rows and 3 columns:

```
⇛ np.zeros((2, 3))
❮ array([[ 0.,  0.,  0.],
         [ 0.,  0.,  0.]])
```

The parameter to `zeros()` has its own parentheses, because it's a *tuple*—an immutable collection of values. In this case, the tuple has two values, that are the rows and columns of the matrix, respectively.

After you initialize the weights to zero, run the network and see what happens. How well does it learn? Try running the zero-initialized network with different numbers of hidden nodes. Do you see a significant difference in accuracy?

Finally, peek into the zero-initialized network as it trains. For example, here is how you can print ten rows from the middle of w_1 and w_2:

The code tag should not be here.

What do you see?

CHAPTER 12

How Classifiers Work

We spent most of this book building classifiers—first a perceptron, and now, in the last few chapters, a full-fledged neural network. And yet, you might struggle to grasp intuitively what makes classifiers tick. Why do perceptrons work well on some datasets, and not on others? What do neural networks have that perceptrons don't? It's hard to answer those questions, because it's hard to paint a mental image of a classifier doing its thing.

The next few pages are all about that mental image. A new concept, called the *decision boundary*, will help us visualize how perceptrons and neural networks see the world. This insight is not only going to nourish your intellect—it will also make it easier for you to build and tune neural networks in the future.

Let's start with a look back at the perceptron.

Tracing a Boundary

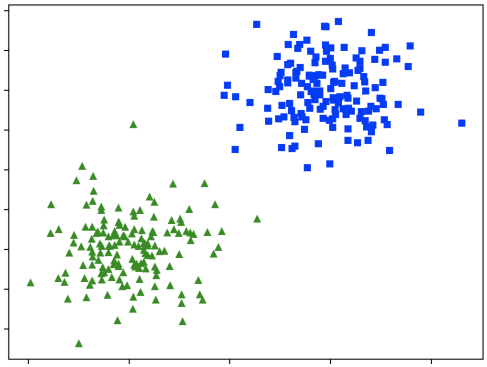
If we hope to understand classification intuitively, then we need a dataset that we can visualize easily. MNIST, with its mind-boggling hundreds of dimensions, is way too complex for that. Instead, we'll use a simpler, brain-friendly dataset:

12_classifiers/linearly_separable.txt

```
Feature_A        Feature_B        Label
-0.470680718301  -1.905835436960  1
0.9952553595720  1.4019246363100  0
-0.903484238413  -1.233058043620  1
-1.775876322450  -0.436802254656  1
```

Those are just the first few lines. The file contains 300 examples in total, each with two input variables and a binary label. Let's plot the data, using the two

input variables as coordinates, and marking each point based on its label—blue squares for 0 and green triangles for 1:



Those are two neatly partitioned classes. We could even trace a straight line between the blue squares and the green triangles. Back in Where Perceptrons Fail, on page 98, we gave a name to datasets that can be partitioned with a line: we called them "linearly separable" datasets. We also said that perceptrons are good at learning linearly separable data. Let's see whether that's true by brushing up the perceptron from Part I of this book.

Browsing this Chapter's Code

The code that generated this chapter's diagrams is in the classifiers directory of this book's downloadable source code. Amongst other things, the classifiers directory includes:

- Two datasets, as text files: linearly_separable.txt and non_linearly_separable.txt.
- The code for our classifiers: the perceptron (perceptron.py) and the neural network (neural_network.py).
- Two programs that load the datasets, train the classifiers, and generate most diagrams: plot_perceptron_boundary.py and plot_neural_network_boundary.py.

Remember, you can download all of the code files used in the book from https://pragprog.com/titles/pplearn/source_code. Happy hacking!

The Perceptron On Its Home Turf

Let's load and format the linearly_separable.txt dataset:

12_classifiers/plot_perceptron_boundary.py

```
import numpy as np
x1, x2, y = np.loadtxt('linearly_separable.txt', skiprows=1, unpack=True)
X_train = X_test = prepend_bias(np.column_stack((x1, x2)))
Y_train_unencoded = Y_test = y.astype(int).reshape(-1, 1)
Y_train = one_hot_encode(Y_train_unencoded)
```

In Training vs. Testing, on page 77, you learned that real-life machine learning systems should use separate data for training and testing. This is not a real-life system, however, so the previous code assigns the same examples to both the training and the test sets. Apart from that detail, it prepares these data pretty much like we did for MNIST: it prepends a bias column to the examples, converts the labels to integers (because `loadtxt()` returns them as floats), and one hot encodes them.

Now we can let the perceptron loose on the data:

12_classifiers/plot_perceptron_boundary.py

```
import perceptron
w = perceptron.train(X_train, Y_train,
                     X_test, Y_test,
                     iterations=10000, lr=0.1)
```

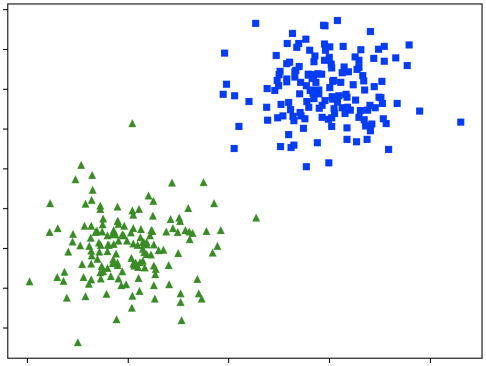
And here's what we get:

```
0 - Loss: 1.38629436111989057245, 50.00%
1 - Loss: 1.29274511217352139347, 100.00%
…
9999 - Loss: 0.00702079217010578415, 100.00%
```

1000 iterations were overkill: it took just one of them for the perceptron to reach perfect accuracy on these data. As we expected, perceptrons eat linearly separable data for breakfast. Let's try to visualize why.

Grokking Classification

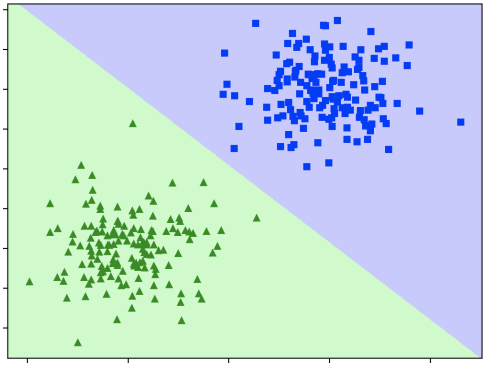
Look at the dataset again:



The idea of classification is: pick any pair of coordinates on the diagram, and the trained classifier will tell you whether it's a blue square or green triangle. Now here's an idea: we could paint each pixel in the diagram, depending on how the perceptron labels it. The code that does that is too long to print here, but here's a pseudocode version of it:

```
# train the perceptron on the dataset
examples = load_dataset()
perceptron.train_on(examples)

# classify and color each pixel
grid = create_grid_overlaying(examples)
for each pixel in grid:
  pixel.color = perceptron.classify(pixel)

# plot the dataset over the colored background
draw(examples)
```

The resulting graph shows the world according to the perceptron:



During training, the perceptron figures out the line that separates the blue and green areas. That line is called the *decision boundary*. During classification, the classifier uses the decision boundary to decide whether a data point is a blue square or green triangle.

The diagram also shows the 300 examples from the training set. They match the areas perfectly: all the blue squares are in the blue area, and all the green triangles are in the green area. That means that if we ask the perceptron to classify one of those points, it will return the correct label. If the perceptron had less than 100% accuracy, then we'd see some blue squares over the green area, or the other way around.

Here is the first crucial piece of information in this chapter: no matter which data and training parameters you use, a perceptron's decision boundary will always be a straight line. Perceptrons don't do curves.

To be clear, I'm talking about a "straight line" because our dataset has two input variables. If it had more, then the boundary would be the higher-dimensional equivalent of a line, like a three-dimensional plane, or a 785-dimensional hyperplane. To be formally accurate, we should say that the decision boundary of a perceptron is a *linear shape*. However, we can just

agree to be informal and say "straight line," because a line is easier to visualize. The concept stays the same with any number of dimensions.

That straight decision boundary is the reason why perceptrons are good at learning linearly separable data. After all, that's definition of "linearly separable": something that can be separated by a straight line. But then, what happens when we train a perceptron on data that's *not* linearly separable?

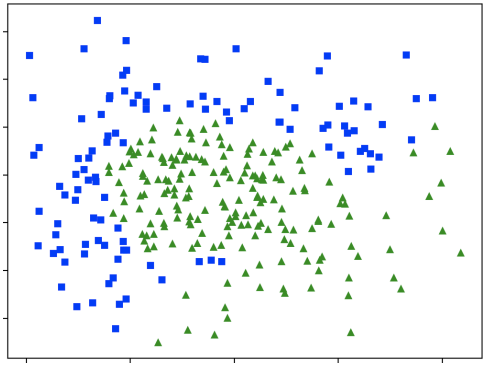
A Line Is Not Enough

Here are a few lines from a second file of data:

12_classifiers/non_linearly_separable.txt

```
Feature_A          Feature_B          Label
0.185872557346525  0.567645715301291  0
0.284770005578376  0.458145376240974  1
0.150041107877389  0.453290078176759  0
```

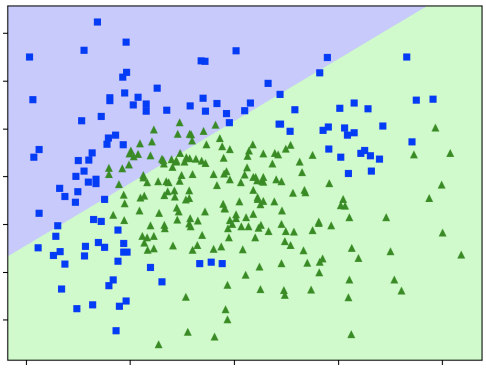
…and so on. This dataset looks the same as the previous one. When we plot it, however, it sings a different song:



There is no way to divide blue squares and green triangles with a straight line. Train the perceptron on these examples, and it will struggle:

```
0 - Loss: 1.38629436111989057245, 36.00%
1 - Loss: 1.38092065346487014033, 64.00%
…
9999 - Loss: 1.09745315956710487448, 73.33%
```

I tried with more iterations and different values of lr, to no avail: I never got anything as high as 80% accuracy. The reason for that weak performance becomes obvious if we plot the perceptron's decision boundary:



See? The perceptron made a valiant attempt at cutting off a slice of blue squares. However, plenty of blue squares are still misclassified as green triangles.

And that's why perceptrons only jive well with linearly separable data. But we're not stuck with perceptrons, are we? Fast forward to our latest and greatest classifier: the neural network.

Bending the Boundary

Let's switch from the perceptron to the neural network. What does its decision boundary look like?

If you bother to try the neural network on the linearly separable dataset, you'll find that it gets the same perfect accuracy as the perceptron, and a similarly straight decision boundary. On the second dataset, however, things get interesting.

Here's the code that trains the neural network on the non-linearly separable dataset:

12_classifiers/plot_neural_network_boundary.py

```
import numpy as np
import neural_network as nn
x1, x2, y = np.loadtxt('non_linearly_separable.txt', skiprows=1, unpack=True)
X_train = X_test = np.column_stack((x1, x2))
Y_train_unencoded = Y_test = y.astype(int).reshape(-1, 1)
Y_train = one_hot_encode(Y_train_unencoded)
w1, w2 = nn.train(X_train, Y_train,
                  X_test, Y_test,
                  n_hidden_nodes=10, iterations=100000, lr=0.3)
```

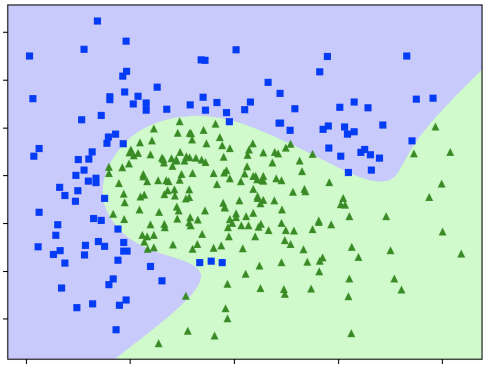
That's pretty much the same code that we used to train the perceptron, with a few minor differences. Note that it doesn't add a bias column to the data, as the network's code takes care of that. As ever, I tried a few different values

for the hyperparameters n_hidden_nodes, iterations, and lr. I ended up with the values above, that seem like a good compromise between training time and accuracy.

Here are the results:

```
Iteration:     0, Loss: 0.65876824, Accuracy: 64.00%
Iteration:     1, Loss: 0.65505330, Accuracy: 64.00%
…
Iteration: 99999, Loss: 0.05917079, Accuracy: 97.00%
```

Now we're talking! And here is the neural network's decision boundary:



With this diagram, we came to the second important point of this chapter: contrary to the decision boundary of a perceptron, the decision boundary of a neural network can be a curve. In fact, our network did a decent job of bending the boundary around the data, leaving just a handful of misclassified points.

Now you know, having seen it with your own eyes, why neural networks leave perceptrons in the dust.

Where the Boundary Comes From

If you have a knack for algebra, then you can see exactly why the decision boundary of a perceptron is a linear shape. (Otherwise, feel free to skip this sidebar. You don't have to understand the math behind the decision boundary, as long as you know what the boundary looks like.)

To help our geometric intuition, let's stick to a dataset with two input variables. We know that the perceptron calculates its output in two steps:

```
weighted_sum = x1 * w1 + x2 * w2 + b
ŷ = sigmoid(weighted_sum)
```

If weighted_sum < 0, then $\hat{y} < 0.5$, which means that the classifier will return a label of 0. Conversely, if weighted_sum > 0, then $\hat{y} > 0.5$, resulting in a label of 1. And if weighted_sum

is exactly 0… well, that's unlikely to happen in practice, so it doesn't really matter which label we get.

Think about the geometric meaning of those formulae. `weighted_sum = x1 * w1 + x2 * w2 + b` is the equation of a plane. `weighted_sum = 0` is also the equation of a plane. So, when we say "the points of the weighted sum that are equal to 0", we're talking about the intersection of two planes, that is a line. That line is the decision boundary. On one side of it, the classifier returns 0; on the other, it returns 1.

Now you have two ways of visualizing the process of training. One is more abstract: "during training, a perceptron descends the gradient of the loss." The other is more geometric: "during training, a perceptron moves a plane in space to define a boundary that separates points with different labels."

Neural networks generate more powerful approximation functions than perceptrons. While a perceptron only generates linear shapes such as planes, a neural networks can concoct a complicated curve. If you slice that curve with a plane, the result is a curvy decision boundary, as we've seen in Bending the Boundary, on page 152.

What You Just Learned

In this chapter we familiarized ourselves with a powerful concept: the *decision boundary* of a classifier. During the training phase, a classifier shapes the decision boundary to separate the classes in the dataset. During the classification phase, it labels data based on which side of the boundary they fall.

Thanks to the boundary, we also understood why a neural network is generally more powerful than a perceptron. A perceptron can only draw a straight decision boundary, so it only works well with *linearly separable* data. A neural network is happy to bend its decision boundary around pretzeled data.

Now we have a better appreciation for neural networks and how they work. And yet, our own neural network didn't do so much better than a perceptron at classifying MNIST. Fear not, because in the next three chapters we're going to tweak and tune the neural network, unleashing its full potential.

Hands On: Data from Hell

In this chapter's source directory you can find an additional dataset named `circles.txt`. Edit the `plot_perceptron_boundary.py` file so that it loads this dataset, and run `python3 data.py` to plot it. Spoiler alert: `circles.txt` is about as non-linearly separable as they get.

Run the perceptron on circles.txt (with python3 plot_perceptron_boundary.py). What does the boundary look like? Try running plot_neural_network_boundary.py. What do you see?

CHAPTER 13

Batchin' Up

By now, we're familiar enough with gradient descent. This chapter introduces a souped-up variant of GD: *mini-batch gradient descent*.

Mini-batch gradient descent is slightly more complicated than plain vanilla GD—but as we're about to see, it also tends to *converge faster*. In simpler terms, mini-batch GD is faster at approaching the minimum loss, speeding up the network's training. As a bonus, it takes less memory, and sometimes it even finds a better loss than regular GD. In fact, after this chapter, you might never use regular GD again!

You might wonder why we're focusing on training speed, when we have more pressing concerns to deal with. In particular, the accuracy of our neural network is still disappointing—better than a perceptron, yes, but well below our target 99% on MNIST. Shouldn't we make the network more accurate first, and faster later? After all, as Donald Knuth has said, "premature optimization is the root of all evil"!

However, there's a reason to speed up training straight away. Within a couple of chapters, we're going to inch toward that 99% goal by tuning the network iteratively: we'll tweak the hyperparameters, train the network… and then do it all over again, until we're happy with the result. Each of those iterations could take hours. We'd better find a way to speed them up—otherwise, the tuning process might take days.

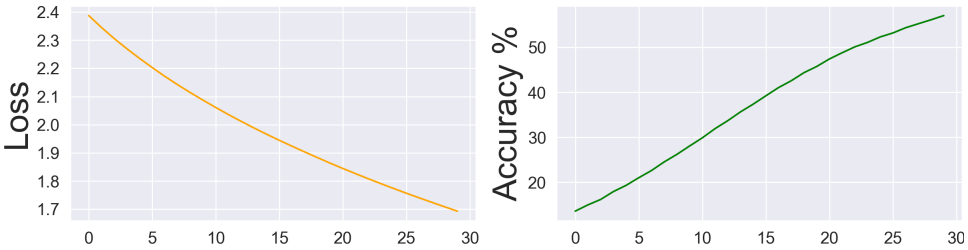
You might also wonder why we're looking for an alternate algorithm, instead of speeding up the algorithm that we already have. Here's the answer: soon enough, we'll stop writing our own backpropagation code, and we'll switch to highly optimized machine learning libraries. Instead of optimizing code that we're going to throw away soon, we'd better focus on techniques that will stay valid even after we switch to libraries.

Mini-batch gradient descent is one of those techniques. Let's see what it's about. But first, let's review what happens when we train a neural network.

Learning, Visualized

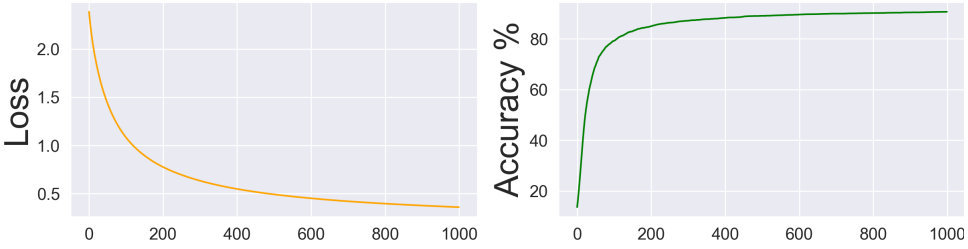
To accelerate training, we need to understand in more detail how it works. Let's take a deeper look at how the neural network's loss and accuracy change during training.

We know that during training, the loss goes down, and the accuracy goes up. Staring at those numbers, however, doesn't tell the whole story. To visualize the loss and accuracy, I hacked the neural network to return two lists—the histories of the loss and the accuracy, stored at each iteration. I trained the network to collect the two histories, and then plotted them over 30 iterations. (If you want to run the code yourself, you can find it in the code download in batching/plot_loss.py.)



These diagrams show what we expect: as we train the network, the loss decreases, and the accuracy increases. However, even looking at the diagrams doesn't really tell us how well the network is doing. For example: we can see that the accuracy is still below 60% after 30 iterations. Is that an early sign of failure, or just the first step toward success? We don't really know until we train longer.

This time, let's train the system for 1000 iterations. We'd better grab something to eat first—on my laptop, this training session takes almost half an hour. Here are the results:



These diagrams are much more informative than the previous diagrams. Now we can see that the loss is *asymptotic*: it drops quickly in the first few iterations, and then approaches an ideal minimum value, without ever quite reaching it. The network's accuracy has a similar shape, only it's pointing upward.

Look closer, and you'll see that the accuracy is never going to reach that target 99%, no matter how long we train this network. In fact, at the end of Chapter 11, Training the Network, on page 127, we trained it for 10000 iterations, and the final accuracy still hovered around 93%. Armed with the previous diagrams, we could have guessed that value even after the first thousand iterations.

To achieve that elusive 99%, we need to tune the network's hyperparameters—and that's where we face the dilemma I mentioned at the beginning of this chapter. On one hand, we must train the network for quite some time just to gauge how well it's doing. On the other hand, we must train it multiple times, so that we can tune it. "Slow" and "frequent" don't mix. It would be nice if the network could give us earlier feedback. Then we could quickly see where that accuracy is aimining, and stop the training if it's not going anywhere.

Can we have our cake, and eat it too? As luck would have it, we can—with a different approach to gradient descent.

Loss vs. Accuracy

If you step though the history of the neural network's loss and accuracy, you might notice one detail. The loss is *strictly decreasing*—it always drops from one iteration to the next. By contrast, the accuracy increases over time, but not necessarily at each iteration. Sometimes it stays stable, and it might even take the occasional turn downwards. Here is why.

The loss decreases at each step because that's what gradient descent does: it steps down the loss's gradient, finding a lower value at each step. In a future chapter, we'll see that the wrong configuration could cause GD to step on a higher loss that it started from—but that's a relatively rare corner case. In general, if we ever see the loss increasing, then we should hasten to change the network's hyperparameters, or debug our GD code.

While the loss is a continuous quantity, the network's accuracy is discrete, because it depends on how many test examples are classified correctly. After the first few minutes of training, that number might stay flat for a long time, while the network fiddles with the weights trying to nail one more example. The accuracy might even drop temporarily, maybe because the network is trading a higher error on a single example with lower errors on many others. For these reasons, while the network's

accuracy does increase in the long term, in the short term it might flatten, or even skid up and down a bit.

To wrap it all up: in gradient descent, the loss normally drops at each and every step. On the other hand, the accuracy climbs—but not necessarily at each and every step.

All that being said, those rules are only valid for plain vanilla GD, and they don't necessarily apply to the GD variants that we're about to discuss. In particular, we're about to abandon the idea that the loss always decreases from one step to the next.

Batch by Batch

The style of gradient descent that we used so far is also called *batch gradient descent*, because it clusters all the training examples into one big batch, and calculates the gradient of the loss over the entire batch. A common alternative is called *mini-batch gradient descent*. Maybe you already guessed what it does: it segments the training set into smaller batches, and then takes a step of gradient descent for each batch.

You might wonder how small batches help speed up training. Stick with me for a moment: let's implement mini-batch GD and give it a test drive.

Implementing Batches

Here is a function that splits the training set into two lists of batches:

```
def prepare_batches(X_train, Y_train, batch_size):
    x_batches = []
    y_batches = []
    n_examples = X_train.shape[0]
    for batch in range(0, n_examples, batch_size):
        batch_end = batch + batch_size
        x_batches.append(X_train[batch:batch_end])
        y_batches.append(Y_train[batch:batch_end])
    return x_batches, y_batches
```

This code loops from 0 to the number of training examples, with a step equal to the batch size. Then it slices `X_train` to get a batch of examples, and it appends the batch to a list. It does the same with `Y_train`. The results are two lists that contain the training set split into batches, where each batch is itself a list.

In case you're wondering, the last batch in the list can be smaller than `batch_size`. If we called the function with a training set of 103 elements and a `batch_size` of 10, then it would happily return two lists of eleven batches, the

last of which only contains three elements. That's okay—mini-batch GD doesn't care whether the batches are all the same size.

Now that we have prepare_batches(), we can update the neural network's report() and train() functions to do mini-batch GD. As usual, look for the arrows in the left margin to spot changes:

13_batching/neural_network.py

```
def report(epoch, batch, X_train, Y_train, X_test, Y_test, w1, w2):
    y_hat, _ = forward(X_train, w1, w2)
    training_loss = loss(Y_train, y_hat)
    classifications = classify(X_test, w1, w2)
    accuracy = np.average(classifications == Y_test) * 100.0
    print("%5d-%d > Loss: %.8f, Accuracy: %.2f%%" %
          (epoch, batch, training_loss, accuracy))

def train(X_train, Y_train, X_test, Y_test, n_hidden_nodes,
          epochs, batch_size, lr):
    n_input_variables = X_train.shape[1]
    n_classes = Y_train.shape[1]

    w1, w2 = initialize_weights(n_input_variables, n_hidden_nodes, n_classes)
    x_batches, y_batches = prepare_batches(X_train, Y_train, batch_size)
    for epoch in range(epochs):
        for batch in range(len(x_batches)):
            y_hat, h = forward(x_batches[batch], w1, w2)
            w1_gradient, w2_gradient = back(x_batches[batch], y_batches[batch],
                                            y_hat, w2, h)
            w1 = w1 - (w1_gradient * lr)
            w2 = w2 - (w2_gradient * lr)
            report(epoch, batch, X_train, Y_train, X_test, Y_test, w1, w2)
    return (w1, w2)
```

There are now two nested loops in train(). The inner loop is a step of gradient descent on a single batch. The outer loop is called an *epoch*, and it goes through all the batches in the training set. Instead of a single hyperpameter named iterations, this version of train() has two: epochs and batch_size.

As a smoke test, let's train the network for two epochs with a batch size of 20000:

```
import mnist as data
w1, w2 = train(data.X_train, data.Y_train,
               data.X_test, data.Y_test,
               n_hidden_nodes=200, epochs=2, batch_size=20000, lr=0.01)
```

Here is the result:

```
0-0 > Loss: 2.38777852, Accuracy: 13.56%
0-1 > Loss: 2.34556340, Accuracy: 14.99%
0-2 > Loss: 2.30576333, Accuracy: 16.17%
```

```
1-0 > Loss: 2.26975858, Accuracy: 17.95%
1-1 > Loss: 2.23549849, Accuracy: 19.46%
1-2 > Loss: 2.20197211, Accuracy: 21.11%
```

The MNIST training set contains 60000 examples, so each epoch spans three batches.

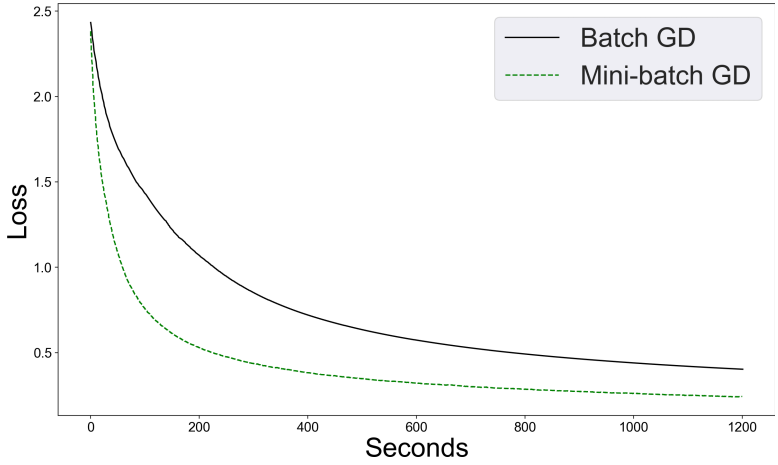
In practice, people often train neural networks with small batches—as small as 16 or 32 elements. Let's see what happens when we use those small batch sizes.

Training with Batches

We're about to perform an experiment, training the neural network on MNIST with two different configurations: plain old batch GD, and mini-batch GD with a batch size of 256. We'll run each training for 20 minutes, and then compare the training history for the two configurations.

To compare the training history, we could use either the history of the network's loss, or the accuracy. Accuracy is what we really care about, but the loss is a more reliable measure, as I discussed in Loss vs. Accuracy, on page 159. That's why people traditionally use the loss for this kind of comparison. We'll do the same.

Here is a diagram of the loss for the two training runs:



Glance at the diagram, and you'll see that mini-batch GD is beating batch GD hands down. Right off the bat, mini-batch GD blazes down the gradient, while batch GD huffs behind. After the first 200 seconds or so, the loss of mini-batch GD is already starting to flatten out, while the loss of batch GD

is still decreasing. After 20 minutes, batch GD is still lagging behind mini-batch GD.

Let me make one thing clear: while mini-batch GD easily wins the speed race, it wouldn't necessarily win the marathon. If we keep the experiment running for hours, then batch GD might eventually catch up. In the short term, however, mini-batch GD is giving us quicker feedback, and that's what we care about while we're tuning the network. With mini-batch GD, it takes a handful of minutes to see how well our network is doing. With batch GD, it takes multiple times as much.

So now you know that mini-batch GD gives us faster feedback. But why?

Understanding Batches

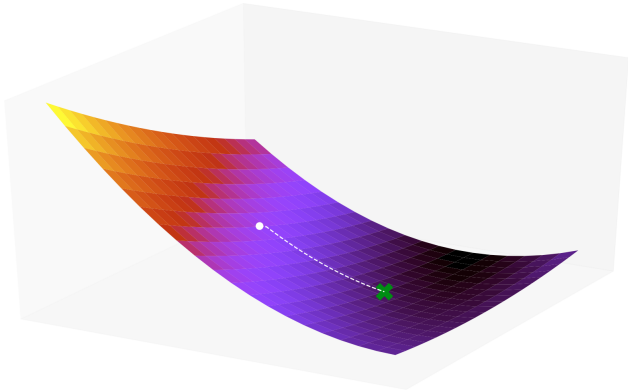
Mini-batch GD feels counter-intuitive. Why do smaller batches result in faster training? The answer is that they don't: if anything, mini-batch GD is generally *slower* than batch GD at processing the whole training set because it calculates the gradient for each batch, rather than once for all the examples.

Even if mini-batch GD is slower, it tends to *converge* faster during the first iterations of training. In other words, mini-batch GD is slower at processing the training set, but it moves quicker toward the target, giving us that fast feedback we need. Let's see how.

Twist that Path

To see why mini-batches converge faster, I visualized gradient descent on a small two-dimensional training set. As usual, you'll find the programs that generate these diagrams amongst this chapter's source code.

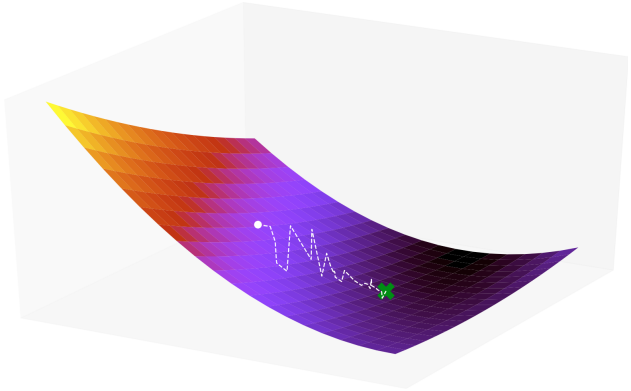
Here is the path of plain-vanilla batch GD during the first few dozens of iterations:



You might remember a similar diagram from back when I introduced gradient descent. The surface is a visualization of the loss. At each iteration, the system calculates the gradient of the loss over the entire training set, and steps in the opposite direction as the gradient, rolling steadily toward the minimum.

Now let's repeat the training using batches. In fact, let's use the smallest possible batch size: a batch size of 1. This extreme variant of mini-batch GD is often called *stochastic gradient descent*, where "stochastic" is statistics lingo for "randomly distributed." The idea of stochastic gradient descent is that you select one random example per iteration, and take a step of GD based on that one example. In our case, we don't even need to select a random example at each iteration because the MNIST dataset has already been shuffled. We can just pick the examples in order, one at a time.

Here is the result of training with stochastic GD:



Now that's an interesting picture. Instead of taking bold steps in the direction of the minimum, stochastic GD staggers toward it like a drunk algorithm, sometimes moving in the *opposite* direction. What's happening?

Here is the reason of that erratic motion: when it comes to calculating the loss, a single example might or might not be representative of the entire training set. For "more typical" examples, the loss surface is similar to the one of the entire dataset, and a step of GD tends to aim straight toward the minimum. On the other hand "less typical" examples result in a pretty different loss surface, that might send GD off in the wrong direction. Because of that uncertain stagger, the global loss does not necessarily decrease at each step, as it does in batch GD.

Even though stochastic GD proceeds in a meandering motion, most steps move toward the general direction of the minimum, and each step happens quickly because it involves a single example. As a result, the algorithm tends to converge quickly, even though it takes a long time to process all the examples.

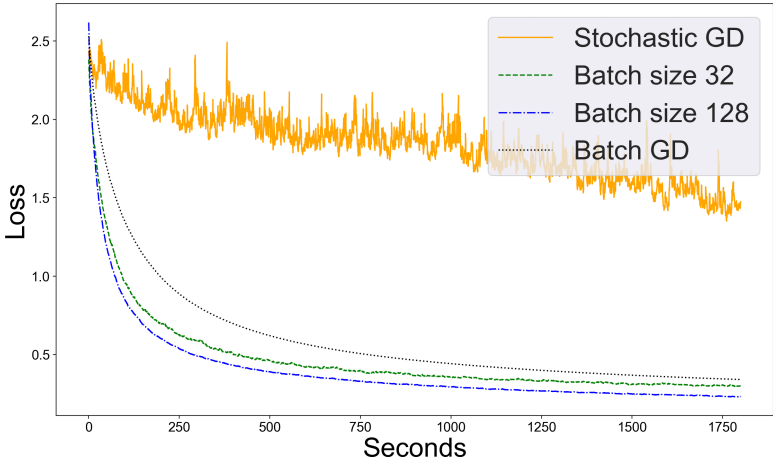
Now let's see what happens if we calculate the loss over more than one example at each step.

Batches Large and Small

This time, we'll train the network for half an hour with four different batch sizes:

- 1 (stochastic GD)
- 32 (mini-batch GD with smaller batches)
- 128 (mini-batch GD with larger batches)
- 60000 (batch GD—one batch for the entire MNIST training set)

I'll spare you the wait. Here are the results:



We can see one thing straight away: stochastic GD might work well for some other problems, but not our particular problem and network configuration. In our case, it results in a loss that jumps up and down like an over-caffeinated grasshopper, and doesn't seem to drop much over time.

As the batch size increases, the loss gets smoother. That happens because the more examples you have, the more likely that their average loss approximates the average loss of the entire training set. As a result, each step of gradient descent is more likely to end up closer to the goal, rather than further away from it.

Smoother, however, is not necessarily better. Batch GD is perfectly smooth, but it doesn't give us the fastest feedback, nor the best final loss. A batch size of 32 gives us faster feedback in the first few iterations, and a lower loss at the end of the experiment—even though batch GD seems likely to catch up in the long term. A batch size of 128 does even better, with great short-term feedback, and a very low final loss. We don't know what happens after 10 or 100 hours of training, but a batch size of 128 looks like a safe bet so far.

And that's what happens when we use different batch sizes on our specific problem. Now let's get the big picture on batches in general.

Batches: The Good and the Bad

I introduced mini-batch GD because it tends to converge faster than batch GD. That benefit, however, is not its only selling point, or even its most important one. Let's go through the other reasons to use batches—and a few reason not to.

Here is the most important benefit of batches: while batch GD forces you to keep the entire training set in memory, mini-batch GD can load data batch by batch, leaving most data offline. If that frugality doesn't sound compelling, consider the astounding size of today's training sets. You can keep MNIST in memory, but you cannot do that with the multi-terabyte training sets that Google and Facebook are using these days. Not only mini-batch GD allows you to load batches one at a time—it also allows you to process those batches in parallel, on multiple cores or servers. Bottom line: as you graduate from small experiments to large systems, mini-batches become your only option.

Mini-batch GD comes with another subtle and important advantage over batch GD. All the way back in When Gradient Descent Fails, on page 44, I mentioned that gradient descent can get stuck into "holes" while stepping

over the gradient surface. I called those holes "local minima," as opposed to the "global minimum"—the lowest point of the loss surface overall.

Just like batch GD, mini-batch GD can fall into a local minimum—but its random fluctuations may get it un-stuck, and back on its way to the global minimum. That's the reason why this variants of GD sometimes results in a lower loss than batch GD, even in the long term. A lower loss generally means more accurate classification, so that's a big deal.

As cool as mini-batch GD is, it does come with a few drawbacks. As we've seen, it requires a bit more code than batch GD. It also introduces `batch_size`, that is yet another hyperparameter that we have to tune. Also, mini-batch GD causes the loss to flutter up and down. If you happen to stop your experiment when the loss is relatively high, then you might lose some accuracy because of sheer bad luck. And finally, very small batch sizes (as in the case of stochastic GD) might be too much of a good thing, and fail to converge, as we've seen in our experiment on MNIST.

Overall, the benefits of mini-batch GD trump its drawbacks. In the next chapters, we'll forget about regular GD and just use batches all the time. If you wish, you can revert to regular GD at any time by setting `batch_size` to the size of the training set.

What You Just Learned

Since I introduced gradient descent, we've been training all of our machine learning systems the same way: for each step of gradient descent, we calculated the gradient of the loss over the entire training set. That flavor of gradient descent is called *batch gradient descent*.

In this chapter, I introduced a different way to do gradient descent: *mini-batch gradient descent*. In mini-batch GD, we take the loss over a *subset* of examples at each step. We also tried an extreme variant of mini-batch GD: *stochastic gradient descent*, where we take the loss on a single example at a time.

Mini-batch GD tends to converge faster than batch GD. As a result, it gives us early feedback on the training. It also tends to be good at escaping "holes" in the loss, so it can yield a lower loss than batch GD. Finally, mini-batch GD is perfect for large training sets that don't fit in memory, and it gives you the option to parallelize training on multiple machines.

Now that you know about mini-batch GD, we're one step closer to tuning our neural network and making it really accurate. We only have one last wrinkle to take care of—and we'll smooth it out in the next chapter.

Hands On: The Smallest Batch

In Batches Large and Small, on page 165, we found out that stochastic GD doesn't work well for our specific problem and neural network configuration. On the other hand, mini-batch GD with a batch size of 32 seems to do okay.

What's the smallest batch size that gives us better early feedback than plain old batch GD? Find out for yourself by changing the batch sizes in `compare_batch_sizes.py`, from the code download.

It is also possible that stochastic GD would work better on our problem if we used different hyperparameters. For example, try it with a smaller learning rate. Do you get a smoother loss?

CHAPTER 14

The Zen of Testing

Soon enough, we'll get to tune our neural network and make it as accurate as we can. Before we do that, however, we need a reliable test to measure that accuracy. As it turns out, ML testing comes with a subtly counterintuitive hurdle that can easily trip you up.

It's hard to explain that hurdle in a few lines—so give me a few pages instead. This short chapter tells you where that testing trapdoor is, and how to step around it.

The Threat of Overfitting

Since Part I of this book, we've been using two separate sets of examples—one for training our algorithms, and one for testing them. Let's refresh our memory: why don't we use the same examples for both training and testing?

I'll use a metaphor to answer that question. Imagine this: you're teaching basic math to a class of young kids. You already prepared 60 multiple-answer multiplication quizzes. You plan to assign most of those quizzes as homework. You also plan to select 10 quizzes for a final test, to check how well the kids are learning.

To recap, you want to split the quizzes in two groups: "homework" and "test." Here are two options: either you assign all 60 quizzes to "homework," and then re-use 10 of them in "test"; or you split the quizzes, assigning 50 of them to "homework," and the remaining 10 to "test." Which option would you pick?

I'd probably go for the second option, for one reason: I don't want the kids to *memorize* multiplications—I want them to *understand the rules* of multiplications. For that reason, I'd rather test their knowledge on quizzes that they haven't seen before. Otherwise, they might cross the right answers just because they happen to remember those answers from their exercises.

The same reasoning applies when we train a supervised learning system. If we train a network to recognize photos of beagles, we want it to recognize *any* beagle picture—not just the specific ones it encountered in training. That's easier said than done. Supervised learning systems, like people, have a tendency to memorize their training data instead of generalizing from it. Back in Training vs. Testing, on page 77, we called this problem "overfitting."

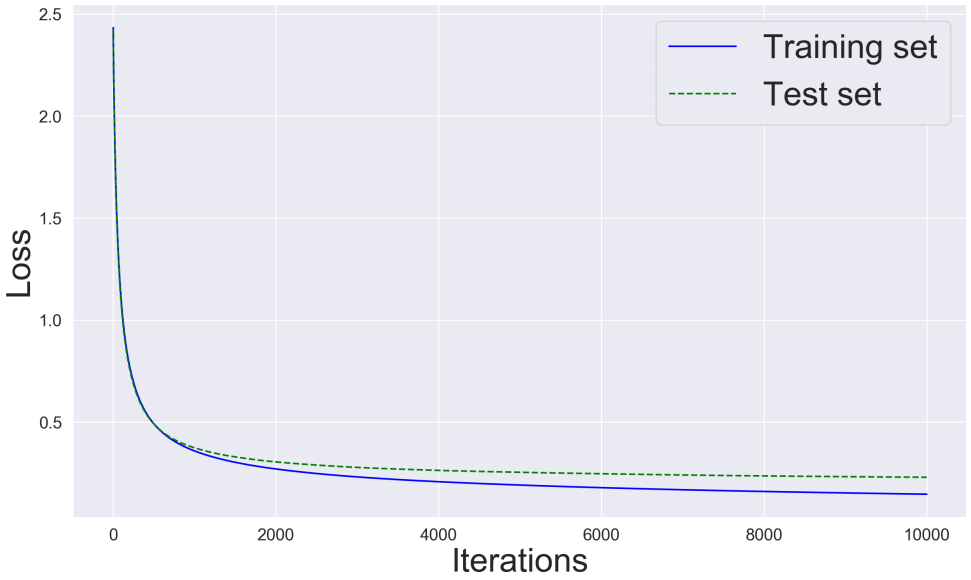
To counter overfitting, we introduced the idea of a test set. Just like the teacher in our story, we train our neural networks on one set of examples, and test them on a different set of examples. That's how we get a realistic estimate of a network's performance in production, where it will be faced with data that it's never seen before.

Because of overfitting, the network yields an unrealistically good performance when it classifies training data, because it's already familiar with those data. We can expect that the network's accuracy on the training set will be higher than its accuracy on the test set. Also, we can expect the opposite result when we compare the system's loss on the two sets: since the loss measures the error in the system's predictions, we can expect a lower loss on the training set than the test set.

Let's put those expectations to the test. I hacked together a version of our neural network that tracks the loss and accuracy on both the training and the test set. Then I ran this hacked network for 10,000 iterations, with batch GD, `n_hidden_nodes=200`, and `lr=0.01`:

```
0 > Training loss: 2.43321 - Test loss: 2.42661
1 > Training loss: 2.38746 - Test loss: 2.38024
2 > Training loss: 2.34527 - Test loss: 2.33774
…
9999 > Training loss: 0.14669 - Test loss: 0.22979
Training accuracy: 96.13%, Test accuracy: 93.25%
```

This chart tracks the loss on both sets:



The two losses start on even ground, but they diverge soon. As the network churns through the training set, its loss on that set decreases faster than the loss on the test set. A lower loss generally means higher accuracy—and indeed, at the end of training the network nails 96% of the training examples, but only 93% of the test examples. If we didn't have a test set, then we'd harbor the illusion that our error rate is below 4%, when in truth it's closer to 7%. That large difference is the effect of overfitting: the network is more accurate on training data, just because—well—that's the data it trained on.

Here is the takeaway lesson—we'll call it "the Blind Test Rule," and capitalize its name to underline that it's important: *test your system on data that it hasn't seen before*. Stick to this rule, and you won't get disappointed by a neural network that's less accurate on real-world data than it was on test data.

"Fair enough," you might say. "We already respect the Blind Test Rule—so, problem solved." Unfortunately, it's easy to violate the rule by mistake. In fact, as we're about to find out, we already did.

A Testing Conundrum

To see where the testing hurdle is, consider that we're going to tune our neural network with an iterative process. That process is going to work like this:

- tune the network's hyperparameters;
- train the network on the training set;

- test the network on the test set;
- repeat until we're happy with the network's accuracy.

The process above is pretty much the ML equivalent of software development, so we can simply call it like that: "the development cycle."

We already went through a few iterations of development in the previous chapter, when we measured the network's performance with different batch sizes. However, we overlooked a distressing fact: the development cycle violates the Blind Test Rule. Here is why.

During development, we tune the neural network's hyperparameters while looking at the network's accuracy on the test set. By doing that, we implicitly custom-taylor the hyperparameters to get a good result on that set. In the end, our hyperparameters are optimized for the test set, and are unlikely to be equally good on never-before seen production data. In a sense, our own brain violates the Blind Test Rule, by leaking information about the test examples into the network. We threw overfitting out of the door, and it sneaked back in through the window.

If you find it hard to understand how the development cycle can cause overfitting, think of a similar example from software development. People commonly use standardized benchmarks to measure the performance of hardware such as graphics cards. Every now and then, a hardware maker is caught "optimizing for the benchmark"—that is, engineering a product to get great results on a particular benchmark, even though those results don't translate as well to real-world problems. The development cycle can easily mislead us into the same kind of "cheating" behavior, where a machine learning algorithm gets unrealistic results on the test set. Call it "unintended optimization," if you wish. To avoid unintended optimization, we shouldn't use the test set during tuning.

Unintended optimization is a sneaky issue. As long as we're aware of it, however, we can avoid it with a low-cost approach: instead of two sets of examples, we can have three—one for training, one for testing, and a third one that we can use during the development cycle. This third set is usually called the *validation set*. If we use the validation set during development, then we can safely use the test set at the very end of the process, to get a realistic estimate of the system's accuracy.

Let me recap how this strategy works:

1. *The setup:* we put the test set aside. We'll never look at it until the very end.

2. *The development cycle:* we train the network on the training set as usual, but we use the *validation* set to gauge its performance.
3. *The final test:* after we've tweaked and tuned our hyperparameters, we test the network on the *test* set, that gives us an objective idea of how it will perform in production.

The key idea in this strategy is worth repeating: put the test set under a stone, and forget about it until the very end of the process. We shouldn't cave in to the temptation of using the test set during the development process. As soon as we do, we'll violate the Blind Test Rule, overfit the test set, and risk unrealistic measures of the system's accuracy.

How many examples should we set aside for the validation and test sets? That depend on the specific problem. Some people recommend setting aside 20% of the examples for the validation set, and just as many for the test set. That's called the "60/20/20" split.

In MNIST, however, we have plenty of examples—70,000 in total. It feels like a waste to set aside almost 30,000 examples for testing. Instead, we can take the 10,000 examples from the current test set, and split them in two groups of 5,000—one for the validation set, and one for the new test set. Here's the updated code that does that:

14_testing/mnist_three_sets.py

```
# X_train/X_validation/X_test: 60K/5K/5K images
# Each image has 784 elements (28 * 28 pixels)
X_train = load_images("../data/mnist/train-images-idx3-ubyte.gz")
X_test_all = load_images("../data/mnist/t10k-images-idx3-ubyte.gz")
X_validation, X_test = np.split(X_test_all, 2)

# 60K labels, each a single digit from 0 to 9
Y_train_unencoded = load_labels("../data/mnist/train-labels-idx1-ubyte.gz")

# Y_train: 60K labels, each consisting of 10 one-hot encoded elements
Y_train = one_hot_encode(Y_train_unencoded)

# Y_validation/Y_test: 5K/5K labels, each a single digit from 0 to 9
Y_test_all = load_labels("../data/mnist/t10k-labels-idx1-ubyte.gz")
Y_validation, Y_test = np.split(Y_test_all, 2)
```

In general, we should take care that the examples are uniformly distributed across the three sets. For example, we wouldn't want the validation set to contain a disproportionate amount of "5" digits, while the test set contains most of the "7" digits. In this case, however, we don't have that problem: the original MNIST test set is already shuffled, so we can just split it in the middle to come up with a validation set and the new test set.

What You Just Learned

So far, we split our examples in a *training set* and a *test set*. This approach, however, tends to break down once we start tuning our system's hyperparameters, because it sneakily leads us to optimize the system for the specific examples in the test set.

In this chapter, we switched to a more sophisticated approach, splitting the test set in two: a smaller test set, and a brand new *validation set*. We'll use the validation set for development, and the test set only once, for our final benchmark. Because that will be the first time that the network comes in contact with the test set, we'll be confident that the results match the network's accuracy on future production data.

One word of warning before we move on: in this chapter, we learned how to measure our system's accuracy correctly by using a test set. However, we didn't eliminate overfitting. Our network is still going to yield unrealistically good results on the training and the validation set—we just decided not to trust those results, and look at the results on the test set instead.

In other words, we worked around overfitting, but we didn't vanquish it. In fact, overfitting is this book's recurring villain, and it will rear its ugly head again. We'll have a final confrontation with overfitting in *Chapter 17, Defeating Overfitting*. For the time being, we'll have to live with it, and neutralize its effects by applying the Blind Test Rule.

Armed with that sound testing strategy, we can finally roll up our sleeves and dive into development.

CHAPTER 15

Let's Do Development

We spent a few chapters building a neural network, and a few more investigating its finer points. In this chapter, we'll come down to the wire and shoot for 99% accuracy on MNIST. To get there, we'll follow an iterative process that is the ML equivalent of software development. In fact, you can call it just that: development.

Like software development, ML development is too broad an activity to fit in this chapter—or this book. It involves people with different skills, from mathematicians to engineers. Even the engineering part of the job is vaster than just "build a neural network and tune it": real-life ML systems are often complicated pipelines composed of multiple algorithms and services. To make things harder, ML development is often an art as well as a science: it requires plenty of experience, educated guesses, and plain old luck.

As the saying goes, however, "the harder you practice, the luckier you get"—so, let's start practicing. We'll look at machine learning development in a nutshell, focusing on three activities:

1. We'll start by preparing data for the network. For example, we'll rescale the input variables to make them more network-friendly.
2. Then we'll move into the development cycle. At each step, we'll improve the network's accuracy by tuning its hyperparameters: `lr`, the batch size, and so on.
3. At the end of the process, we'll put the network to a final test.

Along the way, remember the testing strategy from Chapter 14, The Zen of Testing, on page 169. We have three sets of examples: training, validation, and test. We'll put the test set under a rock right now and ignore it until the final test at the end of the process. Instead, during the development cycle, we'll use the *validation* set to measure the network's performance. In fact, the

validation set is sometimes called the *dev set*, because it's used during development.

Now we have a plan. Let's jump in and see how close we can get to that 99%.

Preparing Data

You might think that a ML engineer spends her time dreaming up and training sophisticated algorithms. Just like programming, however, the job comes with a less glamorous and more time-consuming side. In the case of ML, that grindwork usually involves preparing data.

If you're not convinced that preparing data is a big time sink, think of the effort that went into MNIST. Somebody had to collect and scan 60,000 handwritten digits. They probably hand-checked all those digits to remove the examples that were not representative of real-life digits, maybe because they were too garbled. They also had to center, crop, and scale those images to the same resolution, taking care to avoid graphical artifacts such as jagged edges. I'd wager that they also processed all digits to give them uniform lightness and contrast, from clear-white 0 to pitch-black 255. Last but not least, they labeled each example, and double-checked the labels to sieve out mistakes.

In the case of MNIST, we don't have to do all that work—somebody did it for us. However, we can still massage MNIST a bit further, to make it more friendly to our network.

Preparing data is a complex activity in itself. Here, we're going to scratch its surface by looking at a couple of common techniques, and we'll get an intuitive understanding of what those techniques are for.

Checking the Range of Input Variables

Before you feed data to a neural network, it's a good idea to check that all input variables span similar ranges. Imagine what happens if one input variable ranges from 0 to 10, and another from 1000 to 2000. The two variables might be equally important to predict the label, but the second one would contribute more to the loss, just because it's bigger. As a result, the network would focus on minimizing the loss of the larger variable, and mostly ignore the smaller one.

To avoid that problem, you can rescale the variables to a similar range. That operation is called *feature scaling*, where *feature* is just another name for "input variable." In our case, we don't need to bother with feature scaling,

because all the variables in MNIST are 1-byte pixels—so they never drop below 0 or raise past 255.

However, even if your input variables span a similar range, you don't want that range to extend to large numbers. The problem with feeding large numbers to the network is that they tend to cause large numbers *inside* the network. As we learned in Dead Neurons, on page 139, neural networks work better when they process values that are close to zero, because that's where sigmoids give their best.

Bottom line: if your input variables are spread out (for example, from -10000 to +10000), or off-center (for example, from 10000 to 10100), then you should shift them and scale them to make them small and centered around zero. Let's see a common technique to do that.

Standardizing Input Variables

To keep input variables close to zero, ML practitioners often *standardize* them. "Standardization" means slightly different things to different people, but its most common meaning is this: "take the inputs, subtract their average, and divide them by their standard deviation."

In case you don't know, the *standard deviation* measures how "spread out" a variable is. If the standard deviation is low, that means that the values tend to stay close to their average. For example, the height of humans has a relatively low standard deviation because nobody is hundreds of times taller than anyone else. On the other hand, the height of plants has a high standard deviation because a plant can be as short as moss, or as tall as a redwood.

So, why should you standardize those input variables? Here is what standardization does. If you take a variable and subtract its average, the resulting variable has an average of zero. Similarly, if you take a variable and divide it by its standard deviation, the result has a standard deviation of 1. Apply both operations, and you get the kind of data we want: data that never strays too far from zero.

Here's an updated version of `mnist.py` that standardizes the dataset:

15_development/mnist_standardized.py

```
def standardize(training_set, test_set):
    average = np.average(training_set)
    standard_deviation = np.std(training_set)
    training_set_standardized = (training_set - average) / standard_deviation
    test_set_standardized = (test_set - average) / standard_deviation
    return (training_set_standardized, test_set_standardized)
```

```
# X_train/X_validation/X_test: 60K/5K/5K images
# Each image has 784 elements (28 * 28 pixels)
X_train_raw = load_images("../data/mnist/train-images-idx3-ubyte.gz")
X_test_raw = load_images("../data/mnist/t10k-images-idx3-ubyte.gz")
X_train, X_test_all = standardize(X_train_raw, X_test_raw)
X_validation, X_test = np.split(X_test_all, 2)
```

We could standardize each input variable separately, or all of them together. The input variables in MNIST all have comparable sizes, so the standardize() function standardizes them together. It uses NumPy to calculate the average and standard deviation of the training set, and then applies the formula values - average / standard_deviation.

The last four lines load the two MNIST datasets, standardize them, and split the test data into a validation set and a test set, like we did in the previous chapter. Note that standardize() calculates the average and the standard deviation of the training set—because that's what we care about when we train the network—but then it also standardizes the validation and test sets with the same parameters. That's an important detail because we want the three sets to be similar—otherwise, the network would fail once it moves from training to testing. If we ever deployed our network to production, we'd also have to standardize production data with the same average and standard deviation.

So, we just standardized MNIST. Let's check whether it was worth it.

Standardization in Practice

Let's check the effect of standardization on our neural network's accuracy. The following code runs the network twice, once with regular MNIST and once with standardized MNIST. Each configuration runs for 2 epochs, with a batch size of 60. The MNIST training set contains 60000 examples, so that's 1000 iterations per epoch:

15_development/mnist_vs_standardized_mnist.py

```
import neural_network as nn
import mnist as normal
import mnist_standardized as standardized

print("Regular MNIST:")
nn.train(normal.X_train, normal.Y_train,
         normal.X_validation, normal.Y_validation,
         n_hidden_nodes=200, epochs=2, batch_size=60, lr=0.1)

print("Standardized MNIST:")
nn.train(standardized.X_train, standardized.Y_train,
         standardized.X_validation, standardized.Y_validation,
         n_hidden_nodes=200, epochs=2, batch_size=60, lr=0.1)
```

Here are the results after a few tens of minutes of number crunching:

```
Regular MNIST:
    0-0 > Loss: 2.28073164, Accuracy: 18.90%
    …
    RuntimeWarning: overflow encountered in exp
    …
    1-999 > Loss: 0.41537869, Accuracy: 84.60%
Standardized MNIST:
    0-0 > Loss: 2.26244033, Accuracy: 17.96%
    …
    1-999 > Loss: 0.21364970, Accuracy: 91.80%
```

We don't know what would happen if we kept training the system for days, but this short test gives us pretty compelling numbers. The network is way more accurate when we train it on standardized MNIST than regular MNIST. Also, while regular MNIST causes an overflow during training, standardized MNIST doesn't. It seems that having smaller input variables makes the network more numerically stable, just as the theory goes.

All in all, the verdict is clear: from now on, we'll use the standardized version of MNIST. And now that we have data we trust, let's move into the heart of the development cycle.

Tuning Hyperparameters

We built a neural network and prepared its input data, but that was only the beginning. The same algorithm running on the same data can yield wildly different results, depending on hyperparameters such as the learning rate and the number of hidden nodes.

ML development is mostly about finding good values for those hyperparameters. Compared to software development, that task can look like a form of black magic: there is no hard and fast rule that tells you how to set those hyperparameters. In fact, I chickened out of the issue whenever it came up, offering vague advice like: "try different values for the hyperparameters, and see which ones work better."

In this section, I'll give you some more concrete guidelines. Here's the first: don't change multiple hyperparameters at the same time. Otherwise, you won't know which changes affected the network's accuracy. Instead, let's tune those hyperparameters one at a time, starting with the easy one.

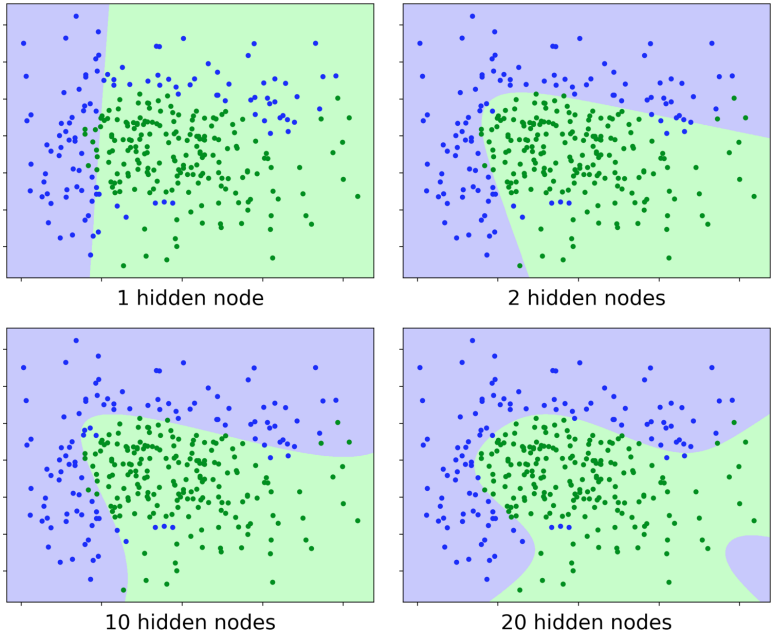
Picking the Number of Epochs

`epochs` is arguably the easiest hyperparameter to tune. We already know the pros and cons of a small versus a large number of epochs: more epochs mean a longer training, but generally better accuracy. At some point, however, the accuracy of the network tends to level off, and training any longer is a waste of time. So, here's the common approach to picking the "right" number of epochs: start with a very high number, and then stop training when the accuracy doesn't seem to increase anymore.

As ever, things are not quite *that* easy. There is some subtleness involved in tuning `epochs`, and in Part III of this book I'll mention cases where the loss *decreases* if you overtrain your network. Those, however, are corner cases. In general, just train as long as you can, and stop when more training doesn't seem to make the network better.

Tuning the Number of Hidden Nodes

Another crucial hyperparameter is the number of hidden nodes that we called `h` in our neural network's code. Here is an experiment that proves how important this hyperparameter is. I trained our network on a simple two-dimensional dataset—the same one that we used in Bending the Boundary, on page 152. Then I plotted the network's decision boundaries for 1, 2, 10, and 20 hidden nodes. Here are the results:



These charts show that, in general, a network with more hidden nodes can draw a more complicated boundary. With a single hidden node, the neural network behaves like a perceptron, with a straight decision boundary. As the number of hidden nodes increases, the network gets better at twisting the boundary and follow the contour of the data.

That doesn't mean we should go wild and use thousands of hidden nodes. First of all, more hidden nodes slow down training. Second, too many hidden nodes can make the network too smart for its own good, leading it to overfit the training data, as I described in The Threat of Overfitting, on page 169. As ever, we've got to find the sweet spot between too few and too many hidden nodes.

Some practitioners use a simple rule of thumb to find a number of hidden nodes that's good enough: they make it the average between the number of input and output nodes. Let's call that the "average rule." Our network has 785 input nodes and 10 output nodes, so the average rule would suggest something around 400 hidden nodes.

compare.py

To compare network configurations throughout this section, I wrote a a simple utility library called compare.py. You use it by calling two functions: configuration() and show_results(). configuration() trains the neural network for a specified time, using a specific set of hyperparameter values. After you've called configuration() a few times, you call show_results() to plot and compare the training histories for all the configurations you've run. You can see a concrete example in Tuning the Number of Hidden Nodes, on page 180.

Be careful when you interpret the results of compare.py. If you train two configurations for a few minutes, and the first yields a better loss, that doesn't mean that it's going to be better in the long term. Good things come to those who wait, and some configurations might simply be slower at pushing down the loss—but once they do, they might push it lower than their speedier competitors. Rather than just look at the final value of the loss, check out the chart plotted by show_results() to get a better idea of what's happening during training. As we said in the beginning of this chapter, configuring a neural network is not an exact science.

Let's test the average rule on our case. We'll use the compare.py utility, which you can read about in compare.py, on page 181. The following code tries a few values of h, both above and below 400:

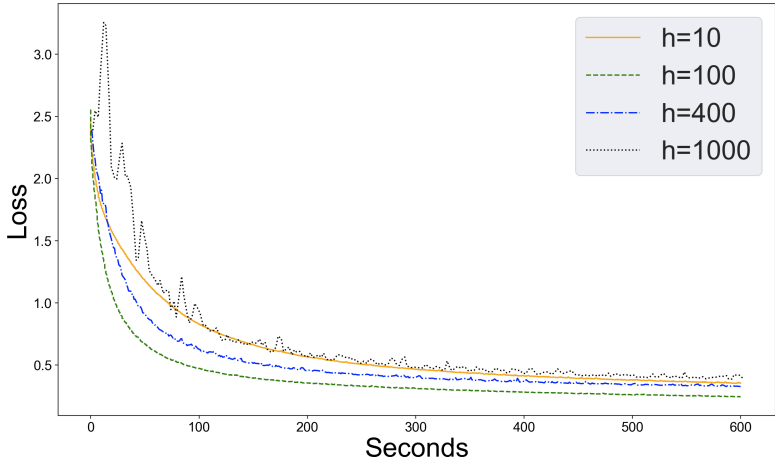
15_development/compare_hidden_nodes.py

```
import compare
import mnist_standardized
```

```
DATA = mnist_standardized
BATCH = 128
LR = 0.1
TIME = 60 * 10
compare.configuration(data=DATA, n_hidden_nodes=10, batch_size=BATCH,
                      lr=LR, time_in_seconds=TIME,
                      label="h=10", color='orange', linestyle='-')
compare.configuration(data=DATA, n_hidden_nodes=100, batch_size=BATCH,
                      lr=LR, time_in_seconds=TIME,
                      label="h=100", color='green', linestyle='--')
compare.configuration(data=DATA, n_hidden_nodes=400, batch_size=BATCH,
                      lr=LR, time_in_seconds=TIME,
                      label="h=400", color='blue', linestyle='-.')
compare.configuration(data=DATA, n_hidden_nodes=1000, batch_size=BATCH,
                      lr=LR, time_in_seconds=TIME,
                      label="h=1000", color='black', linestyle=':')
compare.show_results()
```

Here are the results:

```
Training: h=10
  Loss: 0.35488827 (4 epochs completed, 1932 total steps)
Training: h=100
  Loss: 0.24496126 (2 epochs completed, 1312 total steps)
Training: h=400
  Loss: 0.32815639 (1 epochs completed, 503 total steps)
Training: h=1000
  Loss: 0.39487844 (0 epochs completed, 243 total steps)
```



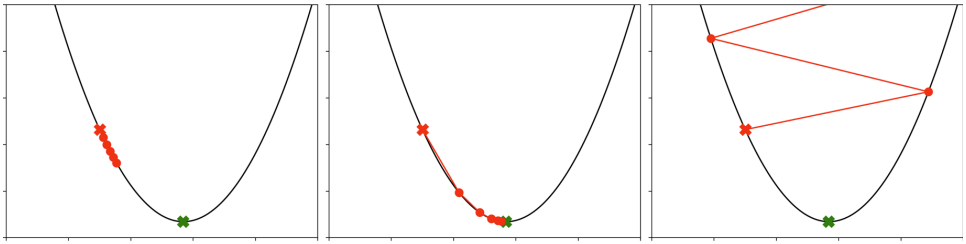
Too many hidden nodes seem to make the training unstable, with the loss jigging a lot. More importantly, 100 hidden nodes result in a lower loss, at least after 10 minutes of training. Ten minutes aren't much, so we might want to review the decision later—but for now, let's settle on 100 hidden nodes.

Number of hidden nodes: checked. Let's move on to the most familiar hyperparameter of them all.

Tuning the Learning Rate

Hello, lr, old buddy. This hyperparameter has been with us since almost the beginning of this book. Chances are, you already had a fling at tuning it, maybe by trying a few random values. It's time to be a tad more precise about lr tuning.

To understand the trade-off of different learning rates, let's go back to the basics and visualize gradient descent. The following diagrams show a few steps of GD along a one-dimensional loss curve, with three different values of lr. The red X marks the starting point, and the green X marks the minimum:



Remember what lr does: the bigger it is, the larger each step of GD is. The first diagram uses a small lr, so the algorithm takes tiny steps towards the minimum. The second example uses a larger lr, which result in bolder steps and a faster descent.

However, we cannot just set a very large lr and blaze towards the minimum at ludicrous speed, as the third diagram proves. In this case, lr is so large that each step of gradient descent lands farther away from the goal than it started. Not only this training process fails to find the minimum, it fails to converge, *increasing* the loss at every step instead of decreasing it. In fact, you could prove mathematically that batch gradient descent always converges, *as long as* lr is sufficiently small. With a large lr, all bets are off.

Now we've seen that a very small lr can slow down GD, and a very large lr can derail GD completely. As ever, we need to strike a balance. Armed with that information, let's brush up compare.py and try a few values of lr:

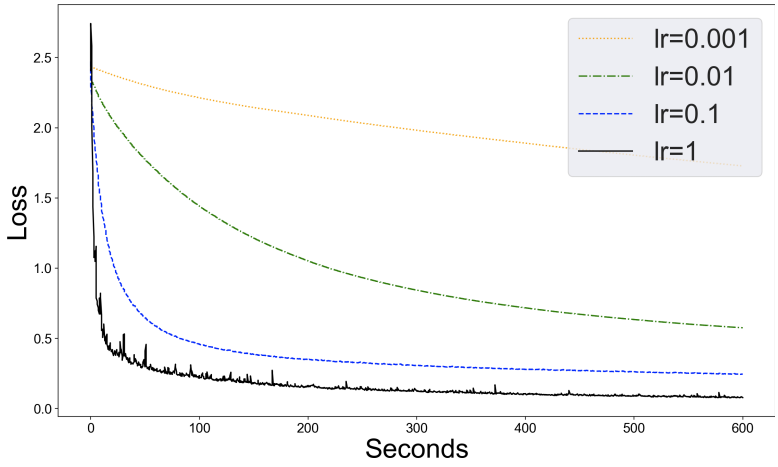
15_development/compare_lr.py

```
import compare
import mnist_standardized

DATA = mnist_standardized
HIDDEN = 100
```

```
BATCH = 128
TIME = 60 * 10
compare.configuration(data=DATA, n_hidden_nodes=HIDDEN, batch_size=BATCH,
                      lr=0.001, time_in_seconds=TIME,
                      label="lr=0.001", color='orange', linestyle=':')
compare.configuration(data=DATA, n_hidden_nodes=HIDDEN, batch_size=BATCH,
                      lr=0.01, time_in_seconds=TIME,
                      label="lr=0.01", color='green', linestyle='-.')
compare.configuration(data=DATA, n_hidden_nodes=HIDDEN, batch_size=BATCH,
                      lr=0.1, time_in_seconds=TIME,
                      label="lr=0.1", color='blue', linestyle='--')
compare.configuration(data=DATA, n_hidden_nodes=HIDDEN, batch_size=BATCH,
                      lr=1, time_in_seconds=TIME,
                      label="lr=1", color='black', linestyle='-')
compare.show_results()
```

lr has a wide range of reasonable values, so it doesn't make sense to try values on a linear scale, such as 0.1, 0.2, and 0.3. Instead, the previous code tries an exponential scale: 0.001, 0.01, 0.1, and 1. In some cases we might have to hunt for even bigger or, more frequently, much smaller values of lr. In our case, it seems that these values span a big enough range:

```
Training: lr=0.001
  Loss: 1.64907297 (3 epochs completed, 1543 total steps)
Training: lr=0.01
  Loss: 0.53636065 (3 epochs completed, 1600 total steps)
Training: lr=0.1
  Loss: 0.23230808 (3 epochs completed, 1552 total steps)
Training: lr=1
  Loss: 0.07311269 (3 epochs completed, 1572 total steps)
```



See those spikes with lr=1? It seems that this value is slightly too large, causing some steps of GD to land further from the minimum than they

started from. On the other hand, it isn't large enough to derail the algorithm entirely, and it yields a lower loss than the other values we tried. Also, after a few minutes of training, it seems that the instability fades away, probably because each step of GD becomes so small that even a large lr isn't enough to make it diverge.

In the long term, it seems that an lr of 1 pays back for its negative effects. We'll stick with this value, and move on to the next hyperparameter.

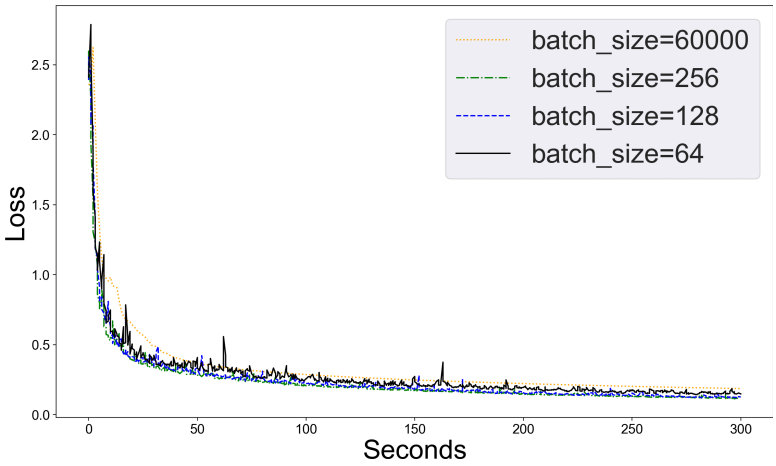
Tuning the Batch Size

We already compared the effects of different batch sizes in Batches Large and Small, on page 165. Back then, we found that batches can speed up training and shorten the development cycle. Now that we're nearing the end of that cycle and have set all the other hyperparameters, let's have another shot at comparing batch sizes:

15_development/compare_batch_sizes.py

```
import compare
import mnist_standardized

DATA = mnist_standardized
HIDDEN = 100
LR = 1
TIME = 60 * 5
compare.configuration(data=DATA, n_hidden_nodes=HIDDEN, batch_size=60000,
                      lr=LR, time_in_seconds=TIME,
                      label="batch_size=60000", color='orange', linestyle=':')
compare.configuration(data=DATA, n_hidden_nodes=HIDDEN, batch_size=256,
                      lr=LR, time_in_seconds=TIME,
                      label="batch_size=256", color='green', linestyle='-.')
compare.configuration(data=DATA, n_hidden_nodes=HIDDEN, batch_size=128,
                      lr=LR, time_in_seconds=TIME,
                      label="batch_size=128", color='blue', linestyle='--')
compare.configuration(data=DATA, n_hidden_nodes=HIDDEN, batch_size=64,
                      lr=LR, time_in_seconds=TIME,
                      label="batch_size=64", color='black', linestyle='-')
compare.show_results()
```

This time, I skipped the test on stochastic GD—that is, a batch size of 1. Last time we tried it, stochastic GD added a lot of noise to our diagram, and it didn't go anywhere in terms of performance. Instead, I focused on three more promising batch sizes: 64, 128, 256, and batch GD—that is, all the examples in one batch. Here is the resulting diagram:



The losses are so close together that we can't see which batch size is doing better in the diagram. For that information, we can look at the results on the terminal:

```
Training: batch_size=60000
  Loss: 0.18655365 (241 epochs completed, 241 total steps)
Training: batch_size=256
  Loss: 0.11773560 (2 epochs completed, 678 total steps)
Training: batch_size=128
  Loss: 0.12600472 (1 epochs completed, 668 total steps)
Training: batch_size=64
  Loss: 0.14859866 (0 epochs completed, 656 total steps)
```

The numbers show that after 5 minutes of training, a batch size of 256 results in a lower loss than the 128 that we used so far. Let's switch to 256 from now on.

That's it. We went through each and every hyperparameter in our neural network, and now we have good values for all of them. Mind you, that doesn't mean that we must stick with those values forever. If we cycled through the hyperparameters a second or a third time, then we'd probably shave a few more decimal points off the loss. Besides, there might be better combinations of hyperparameters—ones that we cannot find by tuning those values one at a time.

All that being said, we did make a lot of progress on tuning our network—so we can call it a day for now. Let's wrap up our short adventure in ML development, and distill all this hard work into one number: our neural network's accuracy.

The Final Test

It's been a while since we reviewed the code of the neural network. Here it is—all of it, for the last time:

15_development/neural_network.py

```
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def softmax(logits):
    exponentials = np.exp(logits)
    return exponentials / np.sum(exponentials, axis=1).reshape(-1, 1)

def sigmoid_gradient(sigmoid):
    return np.multiply(sigmoid, (1 - sigmoid))

def loss(Y, y_hat):
    return -np.sum(Y * np.log(y_hat)) / Y.shape[0]

def prepend_bias(X):
    return np.insert(X, 0, 1, axis=1)

def forward(X, w1, w2):
    h = sigmoid(np.matmul(prepend_bias(X), w1))
    y_hat = softmax(np.matmul(prepend_bias(h), w2))
    return (y_hat, h)

def back(X, Y, y_hat, w2, h):
    w2_gradient = np.matmul(prepend_bias(h).T, (y_hat - Y)) / X.shape[0]
    w1_gradient = np.matmul(prepend_bias(X).T, np.matmul(y_hat - Y, w2[1:].T)
                            * sigmoid_gradient(h)) / X.shape[0]
    return (w1_gradient, w2_gradient)

def classify(X, w1, w2):
    y_hat, _ = forward(X, w1, w2)
    labels = np.argmax(y_hat, axis=1)
    return labels.reshape(-1, 1)

def initialize_weights(n_input_variables, n_hidden_nodes, n_classes):
    w1_rows = n_input_variables + 1
    w1 = np.random.randn(w1_rows, n_hidden_nodes) * np.sqrt(1 / w1_rows)

    w2_rows = n_hidden_nodes + 1
    w2 = np.random.randn(w2_rows, n_classes) * np.sqrt(1 / w2_rows)

    return (w1, w2)

def prepare_batches(X_train, Y_train, batch_size):
    x_batches = []
```

```
    y_batches = []
    n_examples = X_train.shape[0]
    for batch in range(0, n_examples, batch_size):
        batch_end = batch + batch_size
        x_batches.append(X_train[batch:batch_end])
        y_batches.append(Y_train[batch:batch_end])
    return x_batches, y_batches

def report(epoch, batch, X_train, Y_train, X_test, Y_test, w1, w2):
    y_hat, _ = forward(X_train, w1, w2)
    training_loss = loss(Y_train, y_hat)
    classifications = classify(X_test, w1, w2)
    accuracy = np.average(classifications == Y_test) * 100.0
    print("%5d-%d > Loss: %.8f, Accuracy: %.2f%%" %
          (epoch, batch, training_loss, accuracy))

def train(X_train, Y_train, X_test, Y_test, n_hidden_nodes,
          epochs, batch_size, lr):
    n_input_variables = X_train.shape[1]
    n_classes = Y_train.shape[1]

    w1, w2 = initialize_weights(n_input_variables, n_hidden_nodes, n_classes)
    x_batches, y_batches = prepare_batches(X_train, Y_train, batch_size)
    for epoch in range(epochs):
        for batch in range(len(x_batches)):
            y_hat, h = forward(x_batches[batch], w1, w2)
            w1_gradient, w2_gradient = back(x_batches[batch], y_batches[batch],
                                            y_hat, w2, h)
            w1 = w1 - (w1_gradient * lr)
            w2 = w2 - (w2_gradient * lr)
            report(epoch, batch, X_train, Y_train, X_test, Y_test, w1, w2)
    return (w1, w2)
```

Let's put this thing through its final test.

Stretching for 99%

Throughout this chapter, we've been training our network on the training set and measuring its performance on the validation set. As we planned in Chapter 14, The Zen of Testing, on page 169, the time has come to recover the test set that we've been willfully ignoring all this time. We'll classify the entire test set using the hyperparameters we've found in the last few pages:

15_development/final_test.py

```
import neural_network as nn
import mnist_standardized as data

nn.train(data.X_train, data.Y_train, data.X_test, data.Y_test,
         n_hidden_nodes=100, epochs=10, batch_size=256, lr=1)
```

You must be eager to see the results. Here they are:

```
0-0 > Loss: 2.22356892, Accuracy: 26.70%
0-1 > Loss: 2.38862271, Accuracy: 35.50%
0-2 > Loss: 2.08208522, Accuracy: 39.94%
…
9-234 > Loss: 0.04350713, Accuracy: 98.24%
```

After a handful of seconds, the accuracy of the neural network breezes past 90%—and then it just keeps going. On my machine, the network takes a few hours to pass the 98% mark. If you let it run for even longer, it gets up to around 98.6% before leveling up for good. That's not quite the 99% that we aimed for, but it's still very impressive for less than 100 lines of code!

Hands On: Achieving 99%

In the beginning of Part II, we set a target for ourselves: 99% accuracy on MNIST. We just got so close, reaching 98.6%. That last 0.4%? That's up to you.

I told you that configuring a network can be more art than science, and this Hands On is proof of that. Here's my suggestion: to find better hyperparameters than the ones we have now, drop compare.py. Instead, use neural_network_quieter.py, an alternate version of the network that logs accuracy only once every 10 epochs—it's much faster. Here are a few things that you can try:

- The standardized version of MNIST generally works better. There's no reason not to use it.
- Sometimes, it pays off to be patient and wait for more epochs before giving up. However, if 20 or 30 epochs go by without the accuracy increasing at all, then it may be time to try another configuration.
- More hidden nodes slow down the training, but they make the network smarter when dealing with gnarly data. Don't be afraid to push that number higher.
- The smaller the learning rate, the slower the training, but those small step helps the network reach closer to the minimum loss. Try a slightly smaller value of lr than the one we settled on.
- We talked a lot about the pros and cons of small batches in the past. All that being said, in this specific case I had more luck with very large batch sizes.

Hitting that 99% with our neural network is entirely possible. Happy hunting! You can find a solution in ninetynine.py. Feel free to check it if you want to skip

this exercise, or if you're looking to compare your hyperparameters to the ones that I found.

What You Just Learned… and the Road Ahead

Let's consider what we've achieved in this second part of the book. We built a neural network from scratch, we understood how and why it works, and we even worked through advanced details such as mini-batch gradient descent and testing. In this last chapter, we pushed the network as far as we managed, by *standardizing* its input data and tuning its hyperparameters.

Real-world ML development is more complicated than we described in these few pages, just like real-world coding is more complicated than the toy problems in programming books. However, now you have an idea of how to reach the goal that we set for ourselves at the beginning of Part II: get accurate classifications from a neural network. Remember when we thought that 90% accuracy on MNIST was cool? Now we're the proud coders of a 99% accurate network. High five!

We had to work hard to reach that goal, but we never doubted that we would eventually make it. After all, training a neural network sounds like a reasonable task these days. That wasn't always the case, as you might remember from A Tale of Perceptrons, on page 101.

In the 1980s and the 1990s, the battle of ideas between symbolists and connectionists was over, and the symbolists had won. Neural networks were mostly considered a dead end, and only a few researchers were still studying them. Eventually, those researchers implemented backpropagation, and proved the skeptics wrong: it *was* in fact possible to train neural networks! The connectionists had reason to rejoice…and yet, history still seemed to work against them.

You see, neural networks were cool, but also limited. The network that we built in this book is an example of those limitations. Sure, that 1% error rate is impressive—but it cannot compare to the accuracy of a human being, that can identify MNIST characters with close to 99.9% accuracy. What good is a neural network, if a human can do the same job ten times more accurately?

The connectionists believed that neural networks could eventually beat humans—if only they could have more than just three layers. They experimented with so called "deep" neural networks, that had four, five, or even more layers. The results of those experiments were mixed. Deep networks were indeed more accurate than "shallow" three-layer networks, but they were also unstable, finicky, and very hard to train.

Then, in the early 2000s, things changed radically.

In the last part of the book, we'll talk about those changes. Be prepared: if Part I was like crash-landing on an unknown planet, and Part II felt like a space adventure…well, Part III will be like that final sequence in *2001: A Space Odissey*. Let's take a deep breath, and enter deep learning.

Part III

Deep Learning

Deep learning is a set of complex technologies, but they all spring from a simple idea: neural networks become more powerful if they have many layers. Those "deep" networks pose more challenges than the "shallow" three-layer networks that we built so far. In the next few chapters, we'll describe and overcome those challenges.

Adding layers is just the beginning. In the last few years, the original concept of deep learning branched out into many complex—and sometimes wonderful—ideas. We'll take a look at some of those.

In Parts I and II, we set goals for our machine learning systems—respectively, 90% and 99% accuracy on MNIST. In Part III, we'll aim for more ambitious targets. Brace yourself!

CHAPTER 16

A Deeper Kind of Network

This third part of this book is all about *deep learning,* the major breakthrough in modern artificial intelligence. "Deep learning" means a few different things, but first and foremost, it stands for "neural networks with more than three layers."

Later on, you'll see that there is more to deep learning than adding layers. Deep learning is actually a set of interconnected techniques with intimidating names such as "convolutions," "recurrent neural networks," and "generational adversarial networks." By the end of this book, you'll have a better idea of those techniques, and you'll be well equipped to explore them on your own.

To begin, we'll start from that basic concept: adding layers to a neural network. In this chapter, we'll create two networks—a shallow one with three layers, and a deeper one with four. We'll run both networks on the same dataset, and we'll compare their results.

In the first two parts of this book, we proudly wrote our code from scratch, line by line. As you move into deep learning, however, that do-it-yourself approach takes up more and more time. It also becomes less compelling, because you already have a solid grasp of the fundamentals, and don't need to linger on every little detail. Long story short: from this chapter onward, we'll write our neural networks with Keras, a popular ML library. We'll focus on the big picture, and Keras will take care of the nitty-gritties.

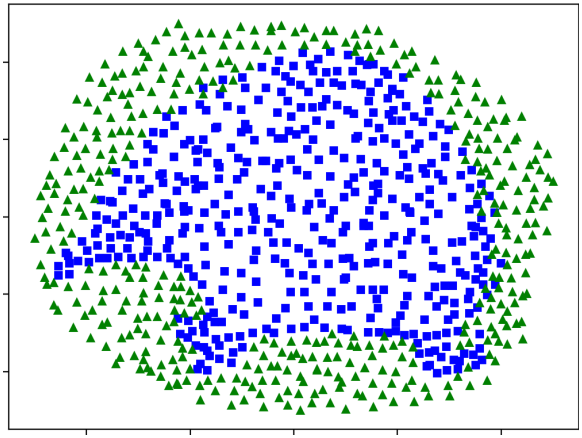
Let's recap what you can expect from this chapter:

1. We'll see how to build a neural network with Keras.
2. We'll write a three-layered network and run it on a simple dataset.
3. We'll add a layer to the network and see how its performance changes.

Before we get into building networks, however, let's prepare a dataset to run them on.

The Echidna Dataset

Do you remember the concept of a decision boundary, introduced back in Tracing a Boundary, on page 147? Soon enough, you'll see how a neural network's decision boundary changes as you add a fourth layer. For that purpose, I wanted a dataset that's twisty, but also easy to visualize. Here it is:



This dataset has two input features and two classes, visualized here as squares and triangles. If you happen to be a fan of marsupials, you might notice that the dataset happens to be shaped like an echidna[1]:

My wife and I still remember an unexpected encounter with a wild echidna on a trip to Australia. Let's not mince words: echidnas are cool.

I saved the Echidna dataset to a file named `echidna.txt`. I also wrote an `echidna.py` file that does the same job as the `mnist.py` file from earlier chapters. It loads

1. https://en.wikipedia.org/wiki/Echidna

the data and the labels into two variables named `X` and `Y`, and also splits it into training, validation, and test sets:

```
⇛ import echidna as data
⇛ data.X.shape
❮ (855, 2)
⇛ data.X[0:3]
❮ array([[ 0.01653543,  0.42533333],
         [ 0.14566929, -0.332     ],
         [ 0.19133858, -0.47866667]])
⇛ data.Y[0:3]
❮ array([[0],
         [1],
         [1]])
⇛ data.X_train.shape
❮ (285, 2)
⇛ data.Y_train.shape
❮ (285, 1)
```

We won't use the test set in this chapter, but the training and validation sets will become useful soon.

Now let's build a neural network that learns the Echidna dataset.

Building a Neural Network with Keras

Python boasts a few large ML libraries such as TensorFlow[2] and Theano.[3] They're complex, and they tend to evolve fast. If I used TensorFlow or Theano in this book, my examples might be obsolete by the time you read them.

Instead, I'll use a slimmer and more stable library named Keras.[4] Keras is a thin layer that sits on top of the "big" libraries and hides them behind a nice, clean programming interface. Keras started out as a tool for prototyping, but it quickly became one of the most popular ways to build neural networks in Python—it was even adopted by TensorFlow as an official interface. With Keras, we can use the heavyweight libraries without getting bogged down by their complexities.

There are different ways to install Keras, depending on your operating system, your CPU and GPU, and so on. On most systems, you can install it like other libraries: either globally via `pip`, or inside a Conda environment. To install globally:

```
pip install keras==2.2.4
```

2. www.tensorflow.org
3. deeplearning.net/software/theano
4. keras.io

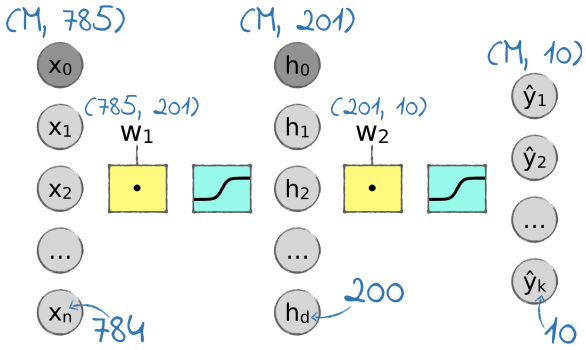
Alternately, here's how you install Keras in the machinelearning Conda environment:

```
source activate machinelearning
conda install keras=2.2.4
```

If you install Keras, you also get TensorFlow—so you can start building neural networks straight away.

A Plan and a Piece of Code

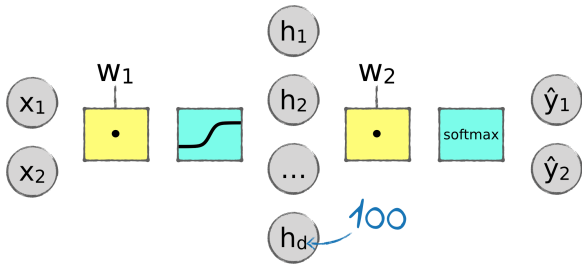
Let's build a neural network for the Echidna dataset. Here is the MNIST network from Chapter 9, Designing the Network, on page 107 that we can use as a starting point:



As a reminder, `M` is the number of examples in the training set, and 784 is the number of features in MNIST. Here is how we can modify this network to learn the Echidna dataset instead:

1. The Echidna dataset has two features, so we only need two input nodes.
2. The Echidna dataset has two classes, so we only need two output nodes instead of 10.
3. The number of hidden nodes is a hyperparameter that we can change later. To begin with, let's go with 100 hidden nodes.
4. Finally, the MNIST network is complicated by the bias nodes x_0 and h_0. We have good news here: Keras takes care of the bias under the hood, so we can forget about the bias nodes altogether.

After applying those changes, here is the plan for our new network:



Now I'm going to show you how you can implement this network with Keras. Don't worry if this code looks confusing—we'll step through it in a minute. Here it is:

16_deeper/network_shallow.py

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import RMSprop
from keras.utils import to_categorical
import echidna as data
import boundary

X_train = data.X_train
X_validation = data.X_validation
Y_train = to_categorical(data.Y_train)
Y_validation = to_categorical(data.Y_validation)

model = Sequential()
model.add(Dense(100, activation='sigmoid'))
model.add(Dense(2, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(lr=0.001),
              metrics=['accuracy'])

model.fit(X_train, Y_train,
          validation_data=(X_validation, Y_validation),
          epochs=30000, batch_size=25)

boundary.show(model, data.X_train, data.Y_train)
```

That isn't much code, considering it's building the network, training it, and even checking its accuracy on the validation set. Let's step through this program line by line.

Loading the Data

These few lines prepare the training set and the validation set:

```
from keras.utils import to_categorical
import echidna as data

X_train = data.X_train
X_validation = data.X_validation
```

```
Y_train = to_categorical(data.Y_train)
Y_validation = to_categorical(data.Y_validation)
```

X_train and X_validation are the same as in echidna.py—the previous code just renames them for consistency. On the other hand, the labels require some more processing because echidna.py doesn't one hot encode them. The last two lines one hot encode the labels with Keras's to_categorical() function, which behaves the same as the one_hot_encode() function we wrote ourselves in Part I. In other words, to_categorical() converts the labels from this…

```
⇛ data.Y_train[0:3]
❮ array([[0],
       [1],
       [1]])
```

…to this:

```
⇛ to_categorical(data.Y_train[0:3])
❮ array([[1., 0.],
       [0., 1.],
       [0., 1.]], dtype=float32)
```

The data is in—now let's assemble the network.

Creating the Model

The next few lines define the shape of the neural network:

```
model = Sequential()
model.add(Dense(100, activation='sigmoid'))
model.add(Dense(2, activation='softmax'))
```

This code uses the object-oriented features of Python. (If you know nothing about objects and classes, then maybe read Creating and Using Objects, on page 266 to get up to speed. It will only take you a few minutes.)

The first line in the code above creates a "sequential model," so called because it assembles a neural network as a sequence of layers. Keras comes with a few options for building neural network, but in this book we'll always use the sequential model.

The second and third line create the hidden layer and the output layer, and add them to the network. There is no need to create an input layer, because Keras takes care of that automatically: as soon as we start feeding data to the network, Keras will look at the shape of the data and create an input layer with a matching number of nodes, which in our case is two. As I mentioned earlier, Keras will also add a bias node to the input and the hidden layer, so we don't need to worry about the bias either.

The model and the layers are Python objects: the model is an object of class `Sequential`, and the layers are objects of class `Dense`. The name "dense" means that the layers are *densely connected*—that is, each node in a layer is connected to all the nodes in the previous layer. We'll look at other types of layers in the next chapters. For the time being, we'll only use densely connected layers.

To create a `Dense` layer, Keras needs two arguments: the number of nodes, and the name of an activation function. Keras supports all the popular activation functions, including the two that we need in this network: the sigmoid and the softmax. Note that for each layer, we specify the activation function that comes *before* the layer, not after it.

We have a neural network. That was quick! Now let's configure it.

Compiling the Model

The next statement configures the neural network—or, in the lingo of Keras, "compiles" it:

```
model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(lr=0.001),
              metrics=['accuracy'])
```

First, this statement tells Keras which formula to use for the loss. We want the same formula we used for the MNIST network in Part II: the cross entropy loss, which Keras calls `categorical_crossentropy`.

Second, this statement tells Keras which algorithm it should use to minimize the loss during training. Keras comes with multiple flavors of gradient descent—in fact, I cheated a bit here: instead of plain vanilla GD (which Keras calls `SGD`), this code uses a souped-up version of GD called `RMSprop`. `RMSprop` is generally better and faster than `SGD`, and that extra speed will be welcome when we start experimenting with this network. I'll go into the details of `RMSprop` later in *Chapter 18, Making It Better.*

When we create the `RMSprop` object, we also pass it the parameters that this particular algorithm needs. Batch GD only needs one parameter: the learning rate `lr`.

Finally, the last parameter to `compile()` tells Keras which metrics to report during training. Keras already reports the loss by default. We also want the accuracy, so we specify that one.

Networking configuration: check. Let's move on to the training phase.

Training the Network

The next statement trains the network—or "fits" it, as Keras prefers to say:

```
model.fit(X_train, Y_train,
          validation_data=(X_validation, Y_validation),
          epochs=30000, batch_size=25)
```

Besides the training set, fit() takes an optional parameter with the validation set. If you pass it the validation set, as in the previous code, Keras will print out your validation loss and accuracy at the end of each epoch, together with the training loss and accuracy. The call to fit() is also where we specify the remaining two hyperparameters: epochs and batch_size.

We're almost done with the neural network's code. We just have one last line to go through.

The Dialects of Machine Learning

One of the challenges of learning a new ML library is getting used to its vocabulary. For example, in these pages, we say that Keras uses the expression "compile the network" to mean "configure the network," and it prefers the term "fitting" to the term "training."

Some of those vocabulary preferences can be quite confusing. For example, Keras uses the name "stochastic gradient descent" (SGD) to mean "GD with batches". By contrast, many people—us included—use that name to mean "GD with a batch size of 1," specifically.

We'll have to live with these slightly confusing names. As you learned by now, the vocabulary of machine learning varies a lot from person to person.

Drawing the Boundary

The last line in the program prints out the neural network's decision boundary. That's not a Keras feature—it's a little utility I wrote called boundary.py that you can find in this chapter's source code. The boundary.show() function takes a trained neural network and a bi-dimensional dataset, and prints out the network's decision boundary over the dataset:

```
import boundary

boundary.show(model, data.X_train, data.Y_train)
```

Note that boundary.show() takes the labels *without* one hot encoding.

With that, we have all it takes to run this neural network on the Echidna dataset, measure its accuracy on the training and the validation set, and check out its decision boundary. Let's run this thing!

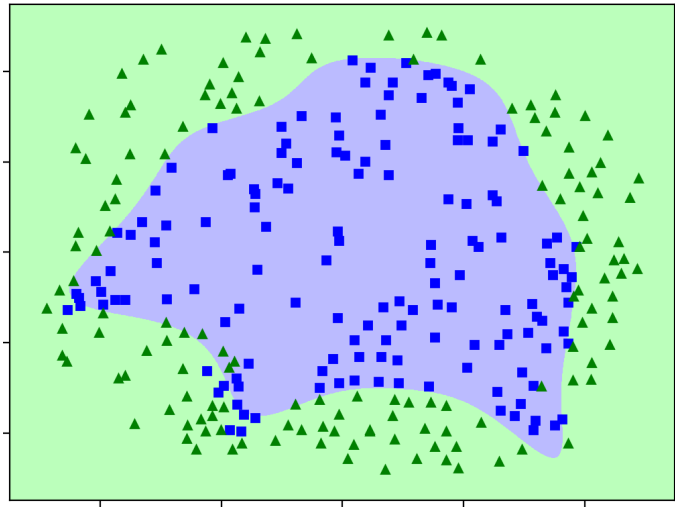
Let's Run It!

Training the neural network took me a few minutes on my laptop. Here's the output, stripped down to the essential information:

```
Using TensorFlow backend.
Train on 285 samples, validate on 285 samples
Epoch 1/30000 - loss: 0.7222 - acc: 0.5088 - val_loss: 0.6928 - val_acc: 0.4807
Epoch 2/30000 - loss: 0.6966 - acc: 0.5018 - val_loss: 0.6904 - val_acc: 0.5193
…
Epoch 30000/30000 - loss: 0.1623 - acc: 0.9193 - val_loss: 0.1975 - val_acc: 0.9018
```

The accuracy on the training set gets up to 0.9193—that is, 91.93%. However, we know from Training vs. Testing, on page 77 that the training accuracy could be polluted by overfitting. Instead, we should look at the validation accuracy: 90.18%.

Now let's check out the network's decision boundary:



The network did a decent job of finding a boundary that separates the data points inside and outside the echidna shape. However, it doesn't seem very good at contouring small details like the echidna's nose and its claws. That lack of finesse in the boundary explains why the neural network misclassifies about one point in ten.

Maybe a deeper network might do better on those small details? After all, that's the selling point of deep learning: just like a shallow neural network

tracks the twists in a dataset better than a perceptron, a deep neural network should do even better.

Let's find out for ourselves, by adding a layer to our neural network.

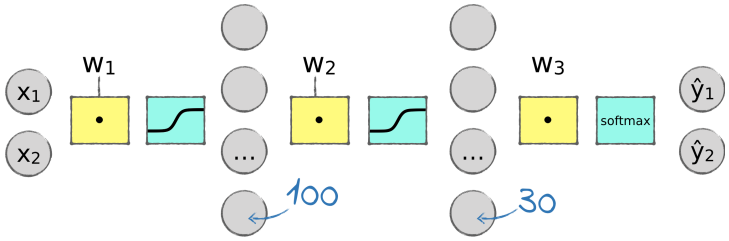
TensorFlow Warnings

Whenever I run Keras-based code, I get a bunch of warnings from the TensorFlow back end. It's telling me that my TensorFlow version and configuration are suboptimal because they don't leverage all the features in my CPU.

I could probably recompile TensorFlow to squeeze some extra performance out of it, but I decided not to. My laptop is never going to be a speed demon anyway, so I'd rather ignore those warnings, and just rent a GPU in the cloud whenever I need serious speed. If you get those warnings as well, you might decide to recompile TensorFlow and gain some speed—but be aware that compiling TensorFlow is a bit of a rabbit hole, so don't expect to be done in a few minutes.

Making It Deep

We just built a three-layered neural network. Now let's give it an additional layer:



This deeper network has four layers, each with its own set of weights and activation function. The number of nodes in the new layer is yet one more hyperparameter, and we should be ready to tune it if we want the best results. To begin with, I set this value at 30.

Let's turn the plan above into code. That's where using Keras really pays off, as adding this layer is a matter of adding one line of code to the model:

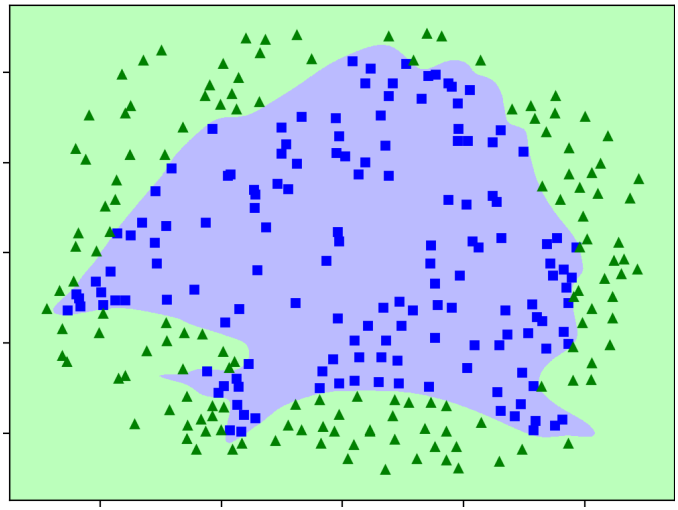
16_deeper/network_deep.py

```
model = Sequential()
model.add(Dense(100, activation='sigmoid'))
➤ model.add(Dense(30, activation='sigmoid'))
model.add(Dense(2, activation='softmax'))
```

That's all we need. Let's run this deeper network and see whether it fares better than the three-layered network we used before:

```
Using TensorFlow backend.
Train on 285 samples, validate on 285 samples
Epoch 1/30000 - loss: 0.7111 - acc: 0.4947 - val_loss: 0.6967 - val_acc: 0.4807
…
Epoch 30000/30000 - loss: 0.0056 - acc: 1.0000 - val_loss: 0.7029 - val_acc: 0.8982
```

And here is the neural network's decision boundary:



Take a moment to look at the numbers and the diagram above. What do you think of them? Do you notice anything specifically?

At a glance, you might think that the network above is amazingly successful. Its decision boundary is exceptionally good at tracking the fine details of the Echidna training set, contouring even the tiniest details to separate squares and triangles with uncanny precision. As a result, the network sports a perfect accuracy of 1.0000 on the training set—which means that it nails the label on 100% of the training examples.

Do a double take, however, and you'll notice something distressing: while this network is perfectly accurate on the training set, it doesn't even come close to that perfection on the validation set. In fact, with 89.92% accuracy, this deep network does *worse* than its shallow counterpart on the validation set!

Let's take a deep breath and consider what we learned from this little experiment.

What You Just Learned

In this chapter, we wrote a quick three-layered neural network with the Keras machine learning library, and we trained that network on a simple dataset.

That shallow network gave us decent results, but it seemed to have trouble dealing with fine details in the data.

We expected that a deeper network would do better, so we added a fourth layer to the network, and trained it on the same dataset. The results were disappointing, to say the least. While the deeper network proved 100% accurate on the training set, it got nowhere near that accuracy on the all-important validation set. In fact, the deeper network did worse on the validation set than the shallow three-layered network.

To recap, we have bad news and good news. First, the bad news: we made our neural network deeper, but that didn't make it better. In fact, it seems that once our network gets deeper, its performance hits a brick wall.

Now, the good news: we can do something about that problem, and unleash the power of deep networks. That's the topic of the next chapter.

Hands On: Keras Playground

Now that you have some Keras under your belt, here is an exercise for you: take the MNIST network we built in Part II of this book and rewrite it from scratch using Keras.

Keras already comes with a few common datasets, MNIST included—so you don't have to use our mnist.py library. Instead, you can load MNIST and one hot encode its labels with this piece of code:

16_deeper/mnist_with_keras.py

```
from keras.datasets import mnist
from keras.utils import to_categorical

(X_train_raw, Y_train_raw), (X_test_raw, Y_test_raw) = mnist.load_data()
X_train = X_train_raw.reshape(X_train_raw.shape[0], -1) / 255
X_test = X_test_raw.reshape(X_test_raw.shape[0], -1) / 255
Y_train = to_categorical(Y_train_raw)
Y_test = to_categorical(Y_test_raw)
```

The first time you run this code, Keras will download MNIST all by itself.

Note that MNIST doesn't have a validation set out of the box. You can either use the test set to validate the network, or split the test set into a validation set and a smaller test set, like we did before.

Note that this code doesn't standardize the inputs as accurately as we did in our earlier implementations: it just divides all the input pixels by 255 so that they'll range from 0 to 1. Feel free to use the more sophisticated standardization method we used in Standardization in Practice, on page 178. When I tried

that method, I found it didn't make a difference for my network—so I chose not to use it.

I showed you the code to load the data, but writing the network is up to you. When it comes time to call `model.compile()`, you can decide which algorithm to use—either `SGD`, that is standard gradient descent, or the more advanced `RMSProp` that we used earlier in this chapter. If you wish, try both and compare the training speed and final accuracy on the training and the test set.

Don't expect that the Keras MNIST network will work well with the same hyperparameters we found in Chapter 15, Let's Do Development, on page 175. You have a different implementation, so you'll probably need to find new values for those hyperparameters. Aim for 99% accuracy or better. If you want to take a peek, my solution is in `mnist_with_keras.py`, in the book's source code.

One last thing: you should notice that the Keras-based neural network is faster than the one we wrote from scratch. That's no surprise—after all, we didn't optimize our code at all. However, most of the CPU power during the training of a neural network is spent multiplying matrices, and our earlier code delegated that operation to the highly optimized NumPy. Bottom line: you should expect the Keras version of the neural network to be faster than the old bespoke version, but not *crazy* faster.

Happy coding!

CHAPTER 17

Defeating Overfitting

Overfitting has been our nemesis throughout the book. In this chapter, we'll finally confront this archenemy.

To refresh your memory, a system that overfits is like a student who learns by rote memory. She might be good at solving familiar problems from textbooks, but she'll struggle when confronted with new problems. Likewise, an overfitting system could be okay at classifying its training data, and then fail when classifying data it hasn't seen before.

In earlier chapters, you learned a strategy to work around overfitting: split your data into training, validation, and test sets. Use the training set to train the system, the validation set to gauge its performance, and the test set for a final check-up. That way, you can test the system on previously unseen data, and get a reliable, overfitting-free measure.

That testing strategy works, but it's a stopgap. It doesn't eliminate overfitting—it just prevents overfitting from polluting our metrics. Unfortunately, overfitting has worse consequences than imprecise metrics. Like that sloppy student I mentioned earlier, an overfitting system is good at *memorizing*, but bad at *generalizing*. We experienced that problem when we built a deep network at the end of the previous chapter. That network reached perfect accuracy on the training set, bit it did worse than its shallow counterpart on the validation set.

In the next few pages, we'll investigate the causes of overfitting and its subtler consequences. Later on, we'll apply a few methods to nip overfitting in the bud—the so-called *regularization* techniques. With those techniques under our belt, we'll finally unleash the power of our deep neural network.

Overfitting Explained

You should know your enemy. So, before we talk about reducing overfitting, let's get to know it better. In this section, we'll dig into the causes of overfitting. What's happening under the hood of an overfitting neural network?

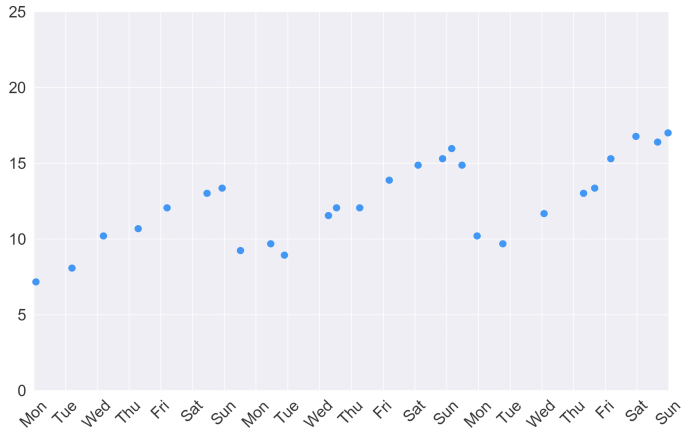
To fully grok overfitting, you should also understand its opposite: *underfitting*. In this section, we'll take a look at underfitting as well.

Let's start by investigating the causes of overfitting.

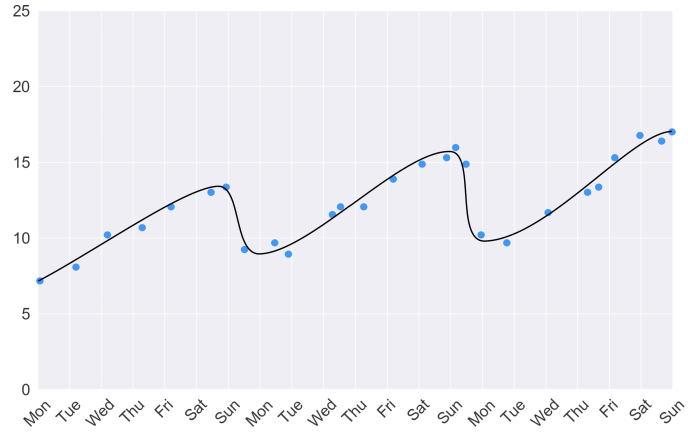
The Causes of Overfitting

So far, I explained overfitting with vague metaphors like "a student memorizing the textbooks." It's time to take a glimpse under the hood of supervised learning and understand how overfitting really happens.

Imagine that we want to predict the number of customers at a hot dog stand, starting from a bunch of historical samples. As usual, we split the samples into training, validation, and test sets. For example, here's a slice of training data that spans the first three weeks of January. The horizontal axis is the day, and the vertical axis is the average number of customers per hour:

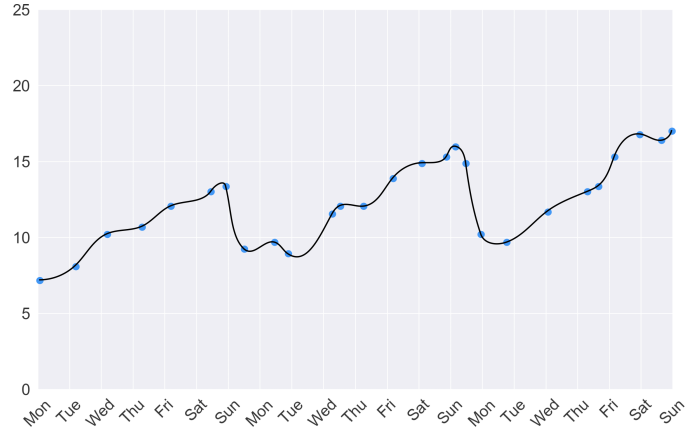


Now consider what happens when we train a neural network with this data. We know from the beginning of this book that supervised learning approximates training data with a model function, like this:



Once the system has this model, it can use it to forecast future customers.

So far, we reviewed how supervised learning works—but here's a catch. If a neural network is powerful enough, it might find a model that fits the data very closely, like this:

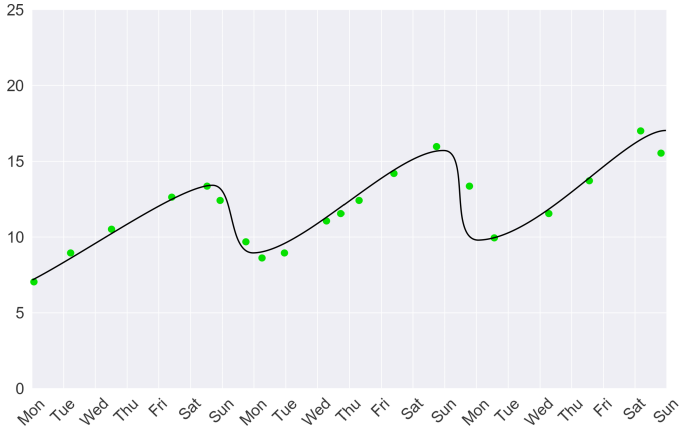


What do you think of this model function? I'd call it too precise for its own good. It tracks every tiny fluctuation in the data, including those that are probably *statistical noise*—irrelevant random variations. For example, our data show an uptick on the second Monday of January. However, we don't expect that the hot dog stand gets extra customers every year on that particular day. It seems more likely that the uptick depended on factors that are irrelevant or unknown. Maybe the weather happened to be especially nice that Monday, so more people where in the streets buying hot dogs.

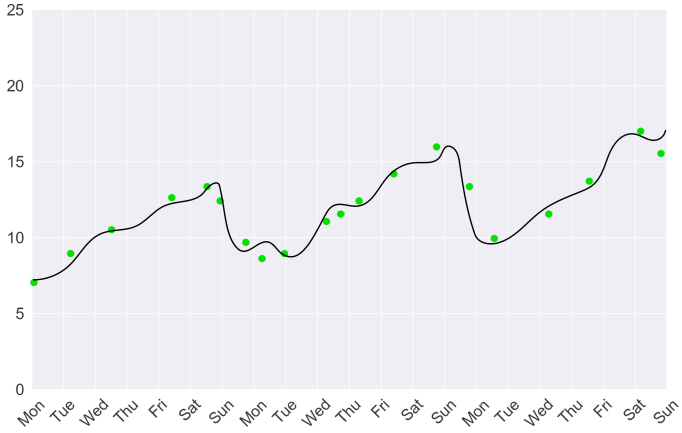
We just came to the root cause of overfitting: it literally means "fitting too well." A powerful neural network can be so eager to fit the data, that it ends up fitting the noise in it ("Yo, there's an uptick on the second Monday of the

year") rather than focus on the meaningful patterns ("Hey, customers tend to grow from Sunday to Saturday").

Now imagine measuring the loss of the two models—the smoother one, and the overfitting one. The overfitting model tracks the data better than the smoother model, so it has a lower loss. That low loss, however, is custom-tailored to the training data. Apply both models to validation data, and the music changes. Here is the smooth model...



...and here is the overfitting model:



With all those pointless nooks and crannies, the overfitting model fits validation data *worse* than the smooth model! That's what overfitting does: at best, it gives you overly optimistic results on training data; at worst, it damages the network's performance on non-training data.

Note that the longer you train this neural network, the more tightly the overfitting model approximates the training data—and the worse it approxi-

mates the validation data. As a result, overfitting can lead to a situation where training becomes counterproductive: more training results in a less accurate network.

I gave you an example based on one-dimensional numerical data, but the same phenomenon happens with higher-dimensional categorical data. All supervised learning systems fit a model to data, so they're all prone to memorize noise. For example, a system that recognizes images of dogs might focus on irrelevant details in the training data ("Look, a yellow pillow!") and lose sight of the important commonalities ("Gee, dogs are hairy"). As it gets better at tracking the noise in the training images, this system gets worse at generalizing to new data.

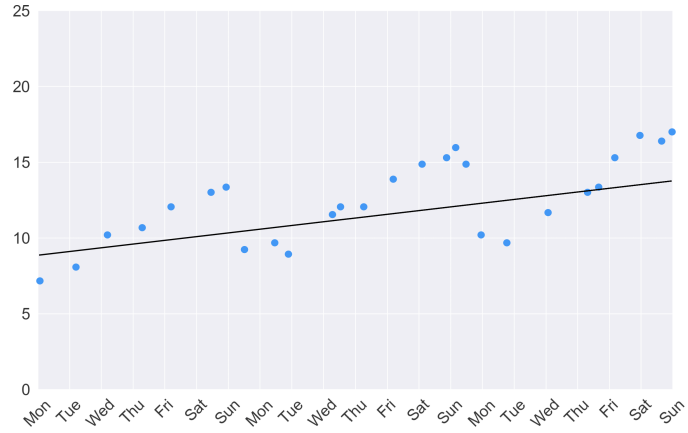
Now you know that powerful neural networks tend to overfit their training data. That's also the reason why I waited until Part III of this book to discuss the details of overfitting. The simple three-layered networks from Part II could hardly overfit complicated datasets like images. On the other hand, the deeper networks that we're building now can overfit even complicated datasets. Once you step into deep learning, that's when overfitting graduates from "minor nuisance" to "pain in the neck."

To wrap it all up, overfitting can cause three problems:

- It confuses our metrics, yielding an unrealistically high performance on the training set.
- It reduces a neural network's performance on unfamiliar data.
- After a certain number of training epochs, it can make additional training counterproductive: the more you train the system, the worse it becomes.

Overfitting vs. Underfitting

You just learned that a powerful system is more likely to overfit its training data. So, what if the system isn't very powerful at all? As an example, consider what happens if you feed the hot dog data to a linear regression program. In that case, the model function might look like this:



This model is too simplistic for the data at hand, so it isn't going to make accurate predictions. You can say that the system is *underfitting*. Just as "overfitting" means "fitting too well," "underfitting" means "not fitting well enough."

In most real-life cases, the data and the model are multi-dimensional, so you cannot plot them like we did in the previous diagrams. However, you can still tell whether your system is overfitting or underfitting by looking at the metrics of loss and accuracy: an overfitting system is good at classifying training data and less good at classifying test data; an underfitting system is bad at both.

You might think that overfitting and underfitting are mutually exclusive: either a system overfits, or it underfits. However, that notion isn't quite correct. In a few pages, we'll see an example of a system that has low accuracy on the training data, but even lower accuracy on the validation data. In other words, a system can overfit and underfit at the same time.

To wrap it up, overfitting and underfitting generally oppose each other: powerful supervised learning systems tend to do the first, and simpler systems tend to do the latter. People sometimes refer to that notion as the *bias-variance tradeoff*, where "bias" and "variance" are alternate terms for "underfitting" and "overfitting" (see Bias and Variance, on page 215). That being said, overfitting and underfitting aren't mutually exclusive, and the same neural network can do a bit of both.

In this section, we stumbled upon one of the major challenges of supervised learning: striking a balance between too much overfitting and too much underfitting. When you build a supervised learning system, you've got to find a middle ground between those two annoying phenomena.

Let's see how to do that, going back to our deep network—and to the Echidna dataset.

Bias and Variance

ML practitioners use the terms "overfitting" and "underfitting" all the time, but statisticians have their own terms for those phenomena. Instead of saying that a model overfits the data, they say that a system has "high variance"; and instead of saying that it underfits the data, they say that it has "high bias."

We won't use the terms "variance" and "bias" in this book, but you might encounter them elsewhere. Just remember that "high variance" stands for "overfitting," and "high bias" stands for "underfitting," and you'll be swell.

Regularizing the Model

You learned three important concepts in the previous section:

- Powerful neural networks tend to *overfit*.
- Simple neural networks tend to *underfit*.
- You should strike a balance between the two.

Here is a general strategy to strike that balance: start with an overfitting model function that tracks tiny fluctuations in the data, and progressively make it smoother until you hit a good middle ground. That idea of smoothing out the model function is called "regularization," and is the subject of this section.

In the previous chapter, we took the first step of the process I just described: we created a deep neural network that overfits the data at hand. Let's take a closer look at that network's model, and afterwards we'll see how to make it smoother.

Reviewing the Deep Network

To gain more insight into overfitting, I made a few changes to our deep neural network from the previous chapter. Here's the updated training code. The rest of the code didn't change, so I skipped it:

17_overfitting/network_deep.py

```
import losses

history = model.fit(X_train, Y_train,
                    validation_data=(X_validation, Y_validation),
                    epochs=30000, batch_size=25)

boundary.show(model, data.X_train, data.Y_train,
```

```
              title="Training set")
boundary.show(model, data.X_validation, data.Y_validation,
              title="Validation set")
losses.plot(history)
```

The original network visualized the decision boundary superimposed over the validation set. I added a second diagram that shows the same boundary over the training set. Besides, I also plotted the history of the losses during training. Conveniently, Keras's model.fit() already returns an object that contains that history—so I just had to write a utility to visualize it, that I called losses.py.

After adding this instrumentation, I trained the deep network again, with these results:

```
loss: 0.0059 - acc: 1.0000 - val_loss: 0.6656 - val_acc: 0.9053
```

Because the weights in the neural network's are initialized randomly, these numbers aren't exactly the same that we got in the previous chapter—but they're in the same ballpark. Once again, we have perfect accuracy on the training set, while the accuracy on the validation set is about 10% lower.

Good thing that we have that validation set! Imagine what could happen without it. Basking in the glory of that illusory 100% accuracy, we'd deploy the network to production... and only then we'd find out that it's way less accurate than we thought.

Data Distribution

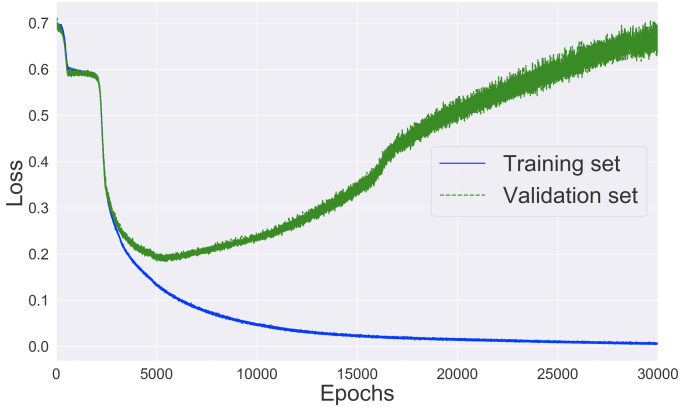
In this book, we're often confronted with a difference between a neural network's accuracy on the training data and on other data (like the validation set, or production data). In all those cases, we explain that difference as an effect of overfitting. However, there might be a simpler reason why a neural network does better on the training set than it does elsewhere. Maybe the examples in the training set are fundamentally different than other data.

For example, consider this scenario: we're building a neural network that recognizes vocal commands, and we train it with thousands of examples recorded by professional speakers. In production, however, the system is confronted with the voices of regular people speaking in noisy environments. The network will have a lower accuracy in production than it had on the training set, but that doesn't happen because of overfitting. It happens because we cheated on your network: we trained it on one kind of data, and ran it on different data. A statistician would say that the training set has a different *distribution* than the production data.

Preparing datasets for a supervised learning system is an advanced topic that's outside the scope of this book. You'll have plenty of time to build up that experience once you start working on real-life projects. Here, we'll just take it for granted that the

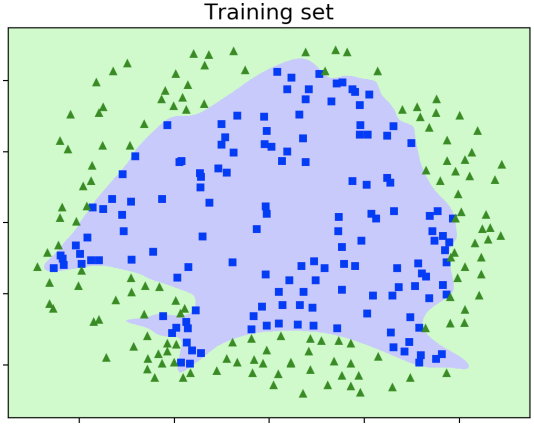
training, validation, and test sets all have the same distribution as the production data.

Those numbers are telling, but the diagrams that follow are even more insightful. Here's the history of the losses during training:

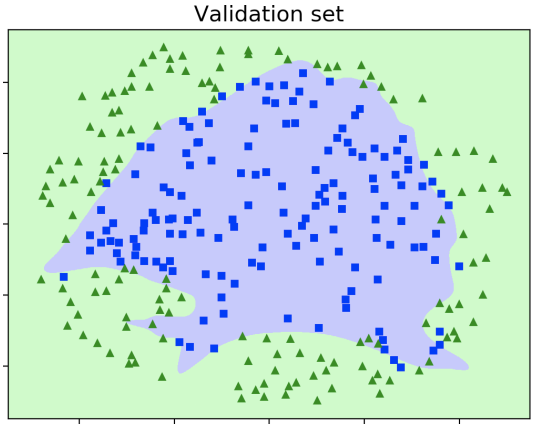


The network starts learning fast, then slows down, then accelerates again—and throughout this process, the losses on the training and the validation set decrease in lockstep. That's until overfitting kicks in, and the two losses part ways. While the training loss keeps happily decreasing, the validation loss skyrockets. That's an obnoxious consequence of overfitting: the longer we train this network, the worse it becomes at classifying new data.

Next, let's check out the network's decision boundary. Here it is, printed over the training set:



This decision boundary expertly contours the training data without missing a single point. When we overlay the boundary on the validation set, however, its limitations become apparent:



Now we see why the network is so inaccurate on the validation set. The ornate convolutions of its decision boundary do more harm than good. This boundary contours the training data closely, but it misclassifies a lot of points in the validation data. In a sense, our deep network is a victim of its own cleverness. A simpler, less baroque boundary would probably do a better job at generalizing to new data points.

We've seen first hand that overfitting may make a network look better, but perform worse. It's time to smooth out that boundary, and improve the neural network's accuracy.

L1 and L2 Regularization

L1 and L2 are two of the most common methods to regularize a decision boundary—the technical name for the operation that we informally called "smoothing out." L1 and L2 work similarly, and they have mostly similar effects. Once you get into advanced ML territory, you may want to look deeper into their relative merits—but for our purposes in this book, I suggest you follow a simple rule: either pick randomly between L1 and L2, or try both and see which one works better.

Let's see how L1 and L2 work.

How L1 and L2 Work

L1 and L2 rely on the same idea: add a regularization term to the neural network's loss. For example, here's the loss augmented by L1 regularization:

$$L_{regularized} = L_{non-regularized} + \lambda \sum |w|$$

In the case of our neural network, the non-regularized loss is the cross entropy loss. To that original loss, L1 adds the sum of the absolute values of all the weights in the network, multiplied by a constant called "lambda" (or λ in symbols).

Lambda is a new hyperparameter that we can use to tune the amount of regularization in the network. The greater lambda, the higher the impact of the regularization term. If lambda is 0, then the entire regularization term becomes 0, and we fall back to a non-regularized neural network.

To understand what the regularization term does to the network, remember that the entire point of training is to minimize the loss. Now that we added that term, the absolute value of the weights has become part of the loss. That means that the gradient descent algorithm will automatically try to keep the weights small, so that the loss can also stay small.

What do small weights have to do with regularization? To answer that question, consider what the weights do: the values in the first layer of the network are multiplied by some of the weights, and then combined together. That process happens again in the second layer, in the third layer, and so on, until the last layer. By consequence, if the weights are big, then a small change of the inputs tends to result in a large change of the outputs. Conversely, if the weights are small, then a small change in the inputs tends to result in a small change of the outputs. In other words, small weights cause the model function to change slowly, instead of jerking up and down.

Bottom line: L1 works by keeping the weights small, and small weights have a regularizing effect on the model function. Bingo! This approach to regularization is also called *weight decay*, because the weights tend to get closer to zero with each step of GD. The L2 method uses the same exact approach, with a minor difference: instead of the absolute values of the weights, it uses their squared values.

Here's a wrap-up of weight decay techniques such as L1 and L2:

- Weight decay techniques roll the weights into the neural network's loss.
- To minimize the loss, GD tends to keeps the weights small.
- Smaller weights tend to result in a smoother model.

Let's use weight decay in our testbed neural network.

L1 in Action

Most machine learning libraries come with ready-to-use weight decay regularization. Let's try L1 regularization in our Keras neural network. First, we need a new import:

```
from keras.regularizers import l1
```

Then we add a new parameter to the network's inner layers:

17_overfitting/network_regularized.py

```
model = Sequential()
model.add(Dense(100, activation='sigmoid', activity_regularizer=l1(0.0004)))
model.add(Dense(30, activation='sigmoid', activity_regularizer=l1(0.0004)))
model.add(Dense(2, activation='softmax'))
```

Keras allows you to set up weight decay for each layer independently. After a few experiments, I regularized both inner layers with $\lambda = 0.0004$, which seems to strike a good balance between overfitting and underfitting. If we wanted to squeeze every last drop of accuracy out of the network, we could tune each layer's lambda separately.

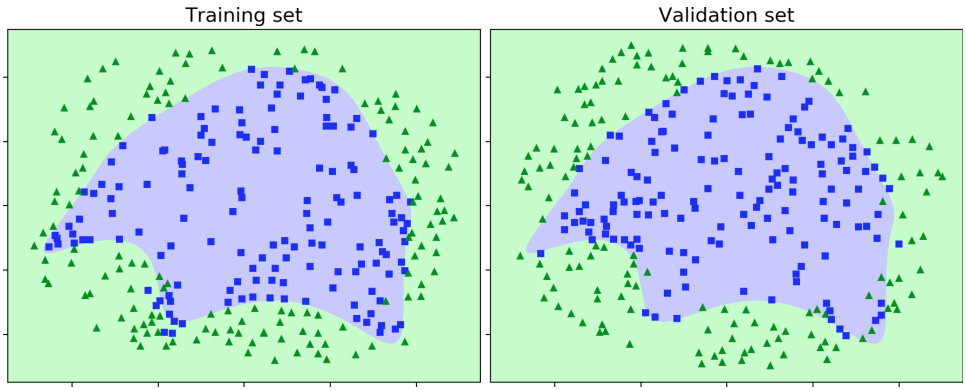
Remember that the non-regularized network gave us about 100% accuracy on the training set, and about 90% on the validation set. By contrast, here's a run of the regularized network:

```
loss: 0.1684 - acc: 0.9509 - val_loss: 0.2278 - val_acc: 0.9263
```

The network's accuracy on the validation set increased from 90% to well above 92%—which means about one fourth fewer errors. That's pretty good! We gave up a 5% of cheap training accuracy for an extra 2% of that precious validation accuracy.

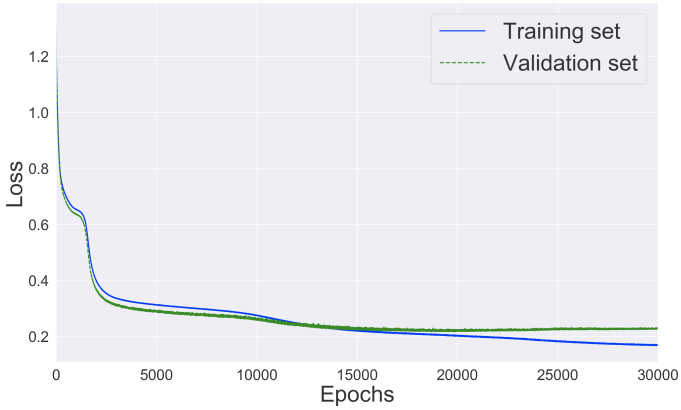
Not only is this network better than the earlier, non-regularized version—it's also better than its shallow counterpart from the previous chapter, that was stuck around 90%. Thanks to regularization, we finally reaped the benefit of having a four-layers network instead of a three-layers network. All this talking about deep learning is starting to win us some concrete progress.

The reason for the neural network's increased performance becomes clear if you look at its decision boundary:



That's a very smooth echidna. This regularized boundary isn't as cleverly convoluted as the one we had before—but it's all the better for that. It doesn't fit the training data quite as well, but it's a better fit for the validation data.

Finally, here's the history of the training loss and the validation loss:



The two losses stay much closer together than before, which means that there is very little overfitting going on.

Too Much of a Good Thing

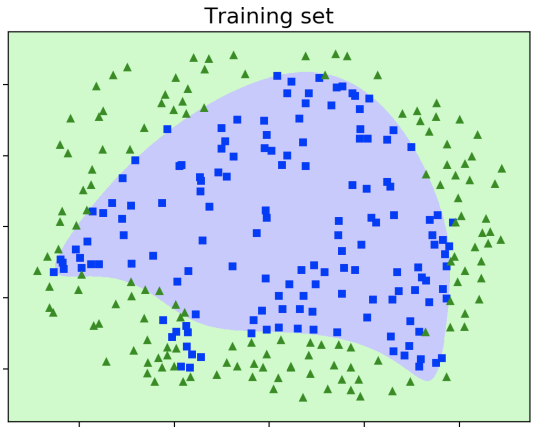
Can't we push lambda higher and decrease overfitting even more? We might, but then we could fall out of the frying pan of overfitting and into the fire of underfitting. For example, here is what I got when I bumped up the lambda of the first hidden layer fivefold, from 0.0004 to 0.002:

```
loss: 0.2392 - acc: 0.9123 - val_loss: 0.2486 - val_acc: 0.9053
```

Our network's accuracy slided back to 90%—no better than it was without regularization, and as much as a shallow, three-layered neural network. This

time, however, the accuracy on the training set took a hit as well. That's a hint that we're now underfitting, not overfitting, the training data.

Take a look at the network's decision boundary, and you'll see where that loss of accuracy comes from:



See? A smoother decision boundary is all well and good, but if you overdo it, the network will lose much of its ability to shape the boundary, and it will underfit the data. That lambda of 0.0004 seems like a good trade-off between overfitting and underfitting for this network and dataset.

With that, we're done talking about weight decay regularization. However, L1 and L2 are far from the only regularization methods we have at our disposal. Let's review a few more before I wrap up this chapter.

A Regularization Toolbox

Just like tuning hyperparameters, reducing overfitting is more art than science. Besides L1 and L2, there are many other regularization methods you can use. Here's an overview of some of them.

The most fundamental regularization technique is also the first one you should reach for: make the overfitting network smaller. After all, overfitting happens because the system is too clever for the data it's learning. Smaller networks are not as clever as big networks. Try reducing the number of hidden nodes, or even removing a few layers. You'll have a go at this approach in the chapter's closing exercise.

Instead of simplifying the model, you can also reduce overfitting by simplifying the input data—that is, removing a few features. Let's say you're predicting a boiler's consumption from a set of 20 input variables. An overfitting network strives to fit the details of that dataset, noise included. Try dropping a few

variables that are less likely to impact consumption (like the day of the week) in favor of the ones that seem more relevant (like the outside temperature). The idea is that the fewer features you have, the less noise you inject into the system.

Here's another way to reduce overfitting: cut short the network's training. This idea is not as weird as it sounds. If you look at the history of the network's loss during training, you can see the system moving from underfitting to overfitting as it learns the noise in the training data. Once overfitting kicks in, the validation flattens, and then diverges from the training loss. If you stop training at that point, you'll get a network that didn't learn enough to overfit the data yet. This technique is called *early stopping*.

Finally, and perhaps surprisingly, sometimes you can reduce overfitting by increasing a neural network's learning rate. To understand why, remember that the learning rate measures the size of each GD step. With the bigger learning rate, GD takes bolder, coarser steps. As a result, the trained model is likely to be less detailed, which might help reduce overfitting.

We went through a few regularization techniques, and we'll see a couple more in the next chapter. Each of these approaches might or might not work for a specific network and dataset. Here, as in many other aspects of ML, your mileage may vary. Be ready to experiment with different approaches, either alone or in combination, and learn by experience which approaches work best in which circumstances.

Collecting More Data

Besides the regularization techniques I describe in these pages, there is another effective approach to reduce overfitting: collect more training data. Intuitively, overfitting happens when a model fails to generalize from its training examples. It's hard to generalize from a handful of examples, and easier if you have plenty of examples. So, the bigger and more varied your training set, the less likely the system is to overfit it.

Collecting data can be expensive, both in terms of time and money. As an alternative approach, you can generate fake training data by modifying the data you already have. For example, if you're training a system to recognize images, you might double the size of your training set just by mirroring your images. Fake data doesn't generally work as well as real data, but it might be the next best thing in some problems.

One last thing: more data helps to reduce overfitting, but it does nothing for a system that's underfitting. It's a common rookie mistake to and fix underfitting with more training data. An underfitting system isn't sophisticated enough to make sense of the data it already has, and collecting more won't help.

What You Just Learned

This chapter was all about *overfitting*. Overfitting happens when the system learns the *statistical noise* in the training data, and fails to generalize that knowledge to new data. The more powerful a supervised learning system is, the more likely it is to overfit. Deep neural networks are very powerful, so they're very prone to overfitting.

You can reduce overfitting by "smoothing out" the neural network's model, so that it follows the general shape of the data instead of tracking every noisy fluctuation. That idea is called *regularization*. In this chapter, we looked at a few regularization techniques:

- *Weight decay* methods (L1 and L2)
- Simplifying the model
- Simplifying the training data
- *Early stopping*
- Increasing the learning rate

To understand overfitting, you should know about the opposite problem: *underfitting*. A network underfits when its model is too simplistic for the data at hand. You recognize overfitting because the system's accuracy is better on the training set than it is on the validation set; you recognize underfitting because the system accuracy is low on both sets.

Overfitting and underfitting aren't mutually exclusive; a network can do a bit of both. However, as you keep reducing the first, you'll eventually increase the second. That catch-22 is known as the *bias-variance tradeoff*. A machine learning expert will aim for the sweet spot between overfitting and underfitting.

And that's it! We confronted overfitting head-on, and came out with a nice collection of regularization methods—and a 2% improvement in accuracy. That, however, is only the beginning. Deep neural networks can take us much farther than that, as long as we get familiar with more, and more varied techniques. In the next chapter, you'll learn enough machine learning tricks to fill up your bag.

Hands On: Keeping It Simple

In this chapter, we applied L1 regularization to reduce overfitting on our four-layers neural network. Now it's up to you to try a few other regularization techniques. How does early stopping work on this network? What about removing a few nodes from each layer?

Try out those techniques, keeping an eye on the accuracy on the validation set. Maybe you'll find a more accurate result than we did in this chapter. Don't worry if you don't! The point of this exercise is experimenting with regularization, not necessarily beating that 92% score.

Once you've optimized the network's accuracy on the validation set, there is one last thing that you can take care of. Do you remember what we talked about in A Testing Conundrum, on page 171? By improving the network's performance on the validation set, we run the risk of overfitting the validation set. There is only one way to find out: recover the Echidna test set, that we've been ignoring until now, and run a final test.

Edit the network's code to replace the validation set (`data.X_validation` and `data.Y_validation`) with the test set (`data.X_test` and `data.Y_test`). Check out the network's accuracy on the test set. Is it as good as the best accuracy you got on the validation set?

CHAPTER 18

Taming Deep Networks

In a sense, there's nothing special about deep networks. They're like three-layered networks, only with more layers. When people started experimenting with deep networks, however, they faced an uncomfortable truth: building deep networks may be easy, but training them is not.

Backpropagation on deep networks comes with its own specific challenges that carry intimidating names such as "vanishing gradients" and "dead neurons." Those challenges rarely come up in shallow neural networks—but they're par for the course in deep neural networks.

Over the years, neural networks researchers developed a collection of techniques, or what I call a "bag of tricks," to tackle those challenges and tame deep neural networks:

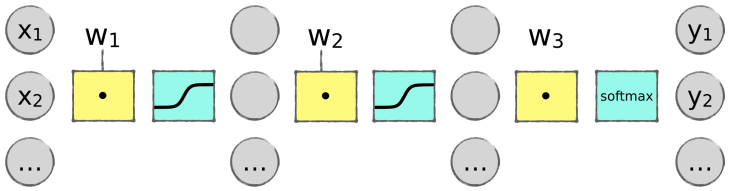
- New activation functions to replace the sigmoid
- Multiple flavors of gradient descent
- More effective weight initializations
- Better regularization techniques to counter overfitting
- Other ideas that work, though they don't quite fit the categories above

This chapter is a whirlwind tour through the list above. We'll spend most of our time discussing activation functions: why the sigmoid doesn't pass muster in deep neural networks, and how to replace it. Then we'll round off the chapter, and your bag of tricks, with a few chosen techniques from the other categories listed here.

Let's start our tour with activation functions.

Understanding Activation Functions

By now, you're familiar with activation functions—those blue boxes in between a neural network's layers:



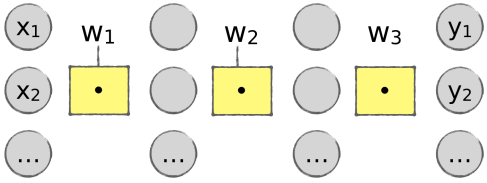
All our activation functions so far have been sigmoids, except in the output layer, where we used the softmax function.

The sigmoid has been with us for a long time. I originally introduced it to squash the output of a perceptron so that it ranged from 0 to 1. Later on, I introduced the softmax to rescale a neural network's outputs so that they added up to 1. By rescaling the outputs, we could interpret them as probabilities, as in: "we have a 0.3 chance that this picture contains a platypus."

Now that we're building deep neural networks, however, those original motivations feel so far away. Activation functions complicate our neural networks, and they don't seem to give us much in exchange. Can't we just get rid of them?

What Activation Functions Are For

Let's see what happens if we remove the activation functions from a neural network. Here is the network we just saw, minus the activation functions:



This network sure looks simpler than the earlier one. However, it comes with a crippling limitation: all its operations are *linear*, meaning that they could be plotted with straight shapes. To explore the consequences of this linearity, let's work through a tiny bit of math.

In a network without activation functions, each of the `n` layers is the weighted sum of the nodes in the previous layer:

$$layer_2 = layer_1 \cdot weight_1$$

$$layer_3 = layer_2 \cdot weight_2$$

...

$$layer_n = layer_{n-1} \cdot weight_{n-1}$$

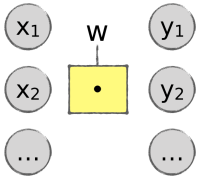
We can squash all those formulae together and calculate the last layer as the dot product of the first layer and all the weights in between:

$$layer_n = layer_1 \cdot weight_1 \cdot weight_2 \cdot \cdots \cdot weight_{n-1}$$

Now, here's the twist: If you take all those dot products of weights and call them `w`, the previous formula boils down to:

$$layer_n = layer_1 \cdot w$$

In other words, we just reduced the entire neural network to a single weighted sum:



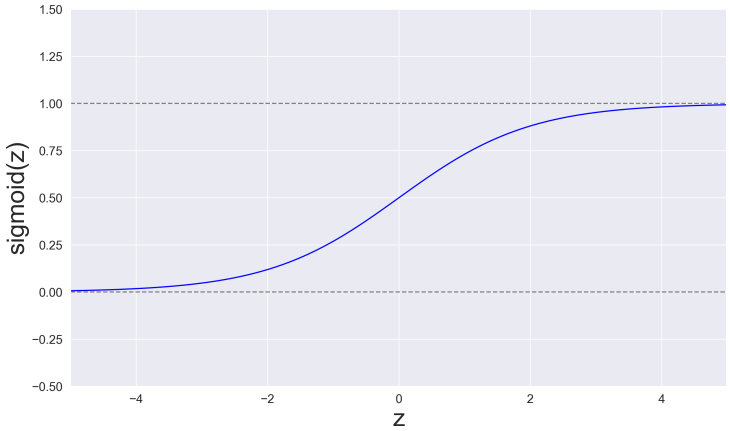
We came to a pretty stark result: No matter how many linear operations we pile up, they always add up to a single linear operation. In concrete terms, a network where everything is linear collapses to a feeble linear regression program, like the one we wrote in Chapter 4, Hyperspace!, on page 47.

Activation functions prevent that collapse because they're *nonlinear*—that is, they cannot be plotted as straight shapes. That nonlinearity is the heart and soul of a neural network.

You just learned that nonlinear activation functions are essential. However, that doesn't mean that we're stuck with the sigmoid and the softmax. We can replace them with other nonlinear functions—and in the case of the sigmoid, in particular, that might be a good idea. Let's see why.

The Sigmoid and Its Consequences

Say hello to our old friend, the sigmoid:



You just learned that an activation function should be nonlinear—that is, non-straight. The curvy sigmoid fits that bill.

At first sight, the sigmoid also jives well with backpropagation. Remember the chain rule? The global gradient of the neural network is the product of the local gradients of its components. The sigmoid contributes a local gradient that's nice and smooth. Slide your finger along its curve, and you'll find no hole, cusp or sudden jump that can trip up gradient descent.

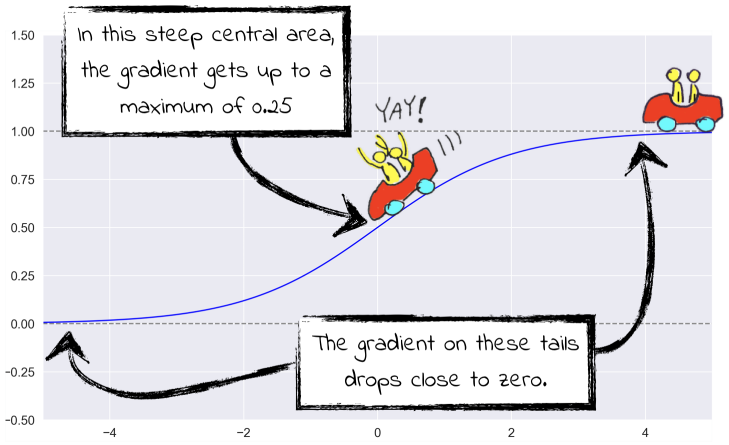
Look closer, however, and the sigmoid begins to reveal its shortcomings. Let's talk about them.

Dead Neurons Revisited

Imagine freezing a neural network mid-training and reading the values in its nodes. Earlier on, you learned an important fact: we have good reasons to want those values small. In Numerical Stability, on page 120, we saw that large numbers can overflow. Then, in Dead Neurons, on page 139, we saw that large numbers can slow down the network's training, and even halt it completely.

If a neural network contains big numbers, those big numbers will eventually pass through a sigmoid. A sigmoid that receives a big positive or negative input can *saturate*—that is, it operates far off-center, on one of its two long tails. On those tails, the gradient is close to zero, and GD grinds to a near halt.

Here's a diagram that illustrates this situation:



The sigmoid's gradient reaches its maximum, that happens to be 0.25, right in the center. As you move along its tails, the gradient drops quickly. If the sigmoid receives a large input, either positive or negative, it enters a vicious circle:

- The larger the sigmoid's input, the smaller the gradient.
- The smaller the gradient, the slower GD.
- The slower GD, the less the network's weights change.
- The less the weights change, the less likely it is the sigmoid's input will eventually shift back toward zero.

When the sigmoid enters the loop above, the nodes uphill of the sigmoid become *dead neurons*. GD is unable to update them, and they stop learning. With too many dead neurons, the whole backprop process slows to a crawl.

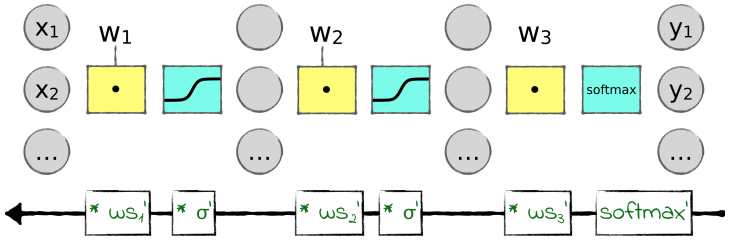
This problem is compound by another subtle issue: the sigmoid is "off-center." When it receives a value that's close to zero, it shifts that value close to 0.5. We said that numbers close to zero are generally better, so it would be nice if the sigmoid didn't go out of its way to push its inputs far away from zero.

To wrap up, the sigmoid only works well when the values in the network are close to zero. That's a though call, especially when the sigmoid itself tends to push values farther from zero. The result is slow backpropagation and dead neurons. The deeper the network, the more sigmoids it contains—and the worse this problem becomes.

Those issues have been known since people became using sigmoids in neural networks. However, the sigmoid is also afflicted by a worse issue—one that only becomes visible in deep networks, and has stumped machine learning researchers for years.

The Vanishing Gradient

Here's our periodic reminder of how backpropagation works: It accumulates the local gradients of all the components in the network from the output to the input layer—like this:



Backpropagation starts from the local gradient of the softmax in the output layer. Then it multiplies it by the gradient of the last weighted sum (that I called ws_3'); it multiplies the result by the gradient of the last sigmoid… and so on, all the way back to the first layer. The gradient of the first layer is the product of the accumulated local gradients.

Now, as you saw in the previous section, the local gradient of a sigmoid is a small number between 0 and 0.25. Whenever backprop traverses a sigmoid, the global gradient is multiplied by that small number, making it smaller. If the network has many layers—and many sigmoids—then the first layers in the network receives a minuscule gradient, and their weights barely change at all.

What I just described is the problem of the *vanishing gradient*. As it moves back through the network, the gradient becomes vanishingly small. This problem puts us in a catch-22: If we want a more powerful network, we should add layers to it; but the more layers we add, the smaller the gradients of the first layers become, to the point where those layers stop learning. We're damned if we add layers, and damned if we don't.

To wrap it all up, we uncovered two significant issues with the sigmoid:

- The problem of *dead neurons*: With a large input, the sigmoid's gradient tends toward zero.
- The problem of *vanishing gradients*: Each sigmoid diminishes the total gradient of the network to the point where the first layers receive a gradient that's close to zero.

Both problems result in tiny gradients that slow down backpropagation, and maybe halt it for good.

Thank you for staying with me! It took us some time to go through this long examination of the sigmoid. On a positive note, that's nothing compared to the time it took researchers to identify some of these problems. For years they struggled to pinpoint why deep neural networks were so hard to train. Once they understood the problems, however, those researchers also came up with solutions. Let's take a look at them.

Don't Play with Explosives

The vanishing gradient has an evil twin called the *exploding gradient* problem. In that case, the gradient's absolute value grows—rather than diminishing—as it backpropagates. Those large numbers in the network cause all sorts of trouble, including dead neurons and overflows.

Exploding gradients happen just like vanishing gradients do: when all the local gradients are multiplied together by the chain rule. If those local gradients are too large, then the network's gradient explodes. That can happen because the weights have been initialized to large values to begin with, or because the network contains activation functions that generate large gradients. For all its shortcomings, the sigmoid is not a suspect in this case: its gradient is always small.

In Better Weight Initialization, on page 238, I'll describe a technique to minimize the risk of exploding gradients.

Beyond the Sigmoid

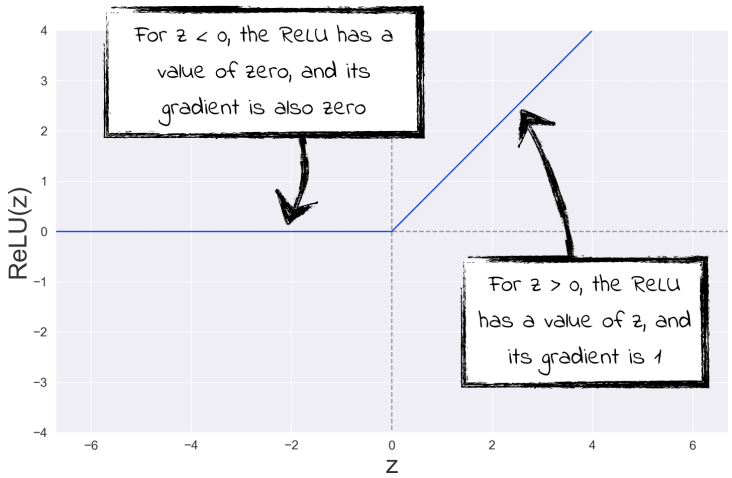
There is no such thing as a perfect replacement for the sigmoid. Different activation functions work well in different circumstances, and researchers keep coming up with brand new ones. That being said, one activation function has proven so broadly useful that it's become a default of sorts. Let's talk about it.

Enter the ReLU

The go-to replacement for the sigmoid these days is the *rectified linear unit*, or *ReLU* for friends. Compared with the sigmoid, the ReLU is surprisingly simple. Here's a Python implementation of it:

```
def relu(z):
  if z <= 0:
    return 0
  else:
    return z
```

And here is what it looks like:



The ReLU is composed of two straight segments. However, taken together they add up to a nonlinear function, like a good activation function should be.

The ReLU may be simple, but it's all the better for it. Computing its gradient is cheap, which makes for fast training. The ReLU's killer feature, however, is that gradient of 1 for positive inputs. When backpropagation passes through a ReLU with a positive input, the global gradient is multiplied by 1, so it doesn't change at all. That detail alone solves the problem of vanishing gradients for good.

ReLUs in Keras

In a Keras neural network, you can replace sigmoids with ReLUs by changing the `activation` parameters from this:

```
model.add(Dense(1200, activation='sigmoid'))
```

…to this:

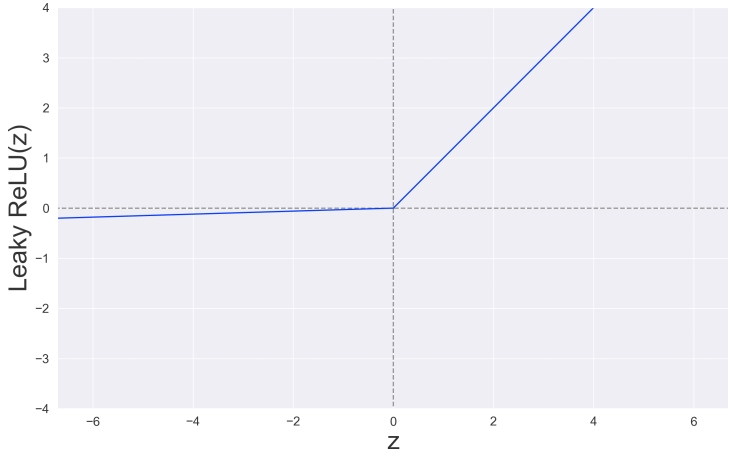
```
model.add(Dense(1200, activation='relu'))
```

That's all you need. Keras will take care of calculating the ReLU's gradient, backpropagating it, and all that jazz. That's a major benefit of libraries: they make it easy to change the design of a neural network on a whim.

While the ReLU gingerly steps around vanishing gradients, it does nothing to solve dead neurons. On the contrary, it seems to have this issue even more than the sigmoid, because it flatlines with negative inputs. On a ReLU with a negative input, GD gets stuck on a gradient of zero, with no hope of ever leaving it. Isn't that a recipe for disaster?

As it turns out, that's not necessarily the case. In some cases, dead ReLUs might even be beneficial to a neural network. Researchers have stumbled on a counterintuitive result: Sometimes, dead ReLUs can make training faster and more effective, because they help the network ignore irrelevant information.

That being said, there is no doubt that too many dead ReLUs can disrupt a neural network. If you suspect that's happening in your own network, you can replace the ReLU with its souped-up sibling, the *Leaky ReLU*, which is illustrated in the following diagram:



Did you spot the difference between the Leaky ReLU and the ReLU? Instead of going flat for negative values, the Leaky ReLU is slightly sloped. That constant slope guarantees that the gradient never becomes zero, and the neurons in the network never die. When you use a Leaky ReLU in a library such as Keras, you can change its slope, so that's one more hyperparameter that you can tune.

Finally, you might have noticed a blemish in the ReLU family. I said that a good activation function should be smooth—otherwise, you cannot calculate its gradient. But the ReLU and the Leaky ReLU aren't smooth: they have a cusp at the value of 0. Won't that cusp trip up GD?

That turns out not to be a problem in practice. From a mathematical standpoint, it's true that the ReLU's gradient is technically undefined at `z = 0`. However, real-life ReLUs are unlikely to ever get an input of exactly 0. Even if that happened, ReLU implementations are only too happy to shrug off mathematical purity, and return a gradient of either 0 or 1 at that point.

Bottom line: it is true that GD might experience a little bump as it transitions through that cusp in the ReLU, but it's not going to derail because of it. If you suspect that the cusp is getting in the way of your training, however, you can Google for variants of ReLU that replace the cusp with a smooth curve. There are at least two: the *softplus* and the *Swish*.

Sigmoid, ReLU, Leaky ReLU, softplus, Swish… You might feel a bit overwhelmed. Don't we have a simple way to decide which of those functions to use in a neural network? Well, not quite—but we can take a look at a few pragmatic guidelines.

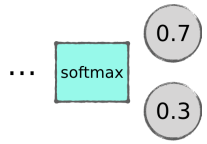
Picking the Right Function

When you design a neural network, you need to decide which activation functions to use. That decision usually boils down to a mix of experience and practical experimentation.

To begin with, however, you can follow a simple default strategy: just use ReLUs. They generally work okay, and they can help you deliver your first prototype as quickly as possible. Later on, you can experiment with other activation functions: Leaky ReLUs, maybe, or perhaps even sigmoids. Those other functions are likely to result in slower training that ReLUs, but they might improve the network's accuracy.

This "ReLU as default" approach applies to all the layers in a network, except for the output layer. In general, the output layer in a classifier should use a softmax to output a probability-like number for each class.

If you only have two classes, then you might want to use a sigmoid instead of a softmax in the last layer. To see why, consider a network that recognizes a vocal command. When you ask the system to classify a sound snippet, a softmax would output two numbers, like this:



This result means "I'm 70% confident that I heard the command, and 30% confident that I didn't." In this case, however, the second output isn't very useful because it always equals 1 minus the first output. For that reason, the softmax is overkill with only two classes, and you can replace it with a sigmoid:

We took a long tour through activation functions, and we walked out with a few useful guidelines. Now let's look at a few other techniques that can help you tame deep neural networks.

Predicting Scalars

The examples in this section, as in most of the book, assume we're building a classifier. However, many neural networks are not classifiers, and their output is scalar rather than categorical. For example, a network might forecast the temperature in a vat, or the number of pizzas sold at a restaurant. (The last example should ring a bell.)

Networks with a scalar output have a few differences from classifiers:

- They have one node in the output layer, instead of one node per class.
- Instead of the cross entropy loss, which only works for categorical outputs, they use a loss that works for scalars, such as the mean squared error.
- Finally, and particularly relevant to this chapter, if the network's outputs is scalar, then you don't need an activation function in the output layer. That lonely node node in the output layer—that's the output of the network.

For consistency, instead of skipping the activation function in the output layer, you can use an identity function. The identity function returns its input, so it's the same as not having an activation function at all. This is a rare case of a linear, rather than non-linear, activation function.

Adding More Tricks to Your Bag

Picking the right activation functions is a crucial decision, but when you design a neural network, you face plenty more. You decide how to initialize the weights, which GD algorithm to use, what kind of regularization to apply, and so forth. You have a wide range of techniques to choose from, and new ones come up all the time.

It would be pointless to go into too much detail about all the popular techniques available today. You could fill an entire book with them. Besides, some of them might be old-fashioned and quaint by the time you read this book.

For those reasons, this section doesn't aspire to be comprehensive. See it like a quick shopping spree in the mall of modern neural networks: we'll look at a handful of techniques that generally work well—a starter's kit in your

journey to ML mastery. At the end of this chapter, you'll also get a chance to test these techniques first hand.

Let's start with weight initialization.

Better Weight Initialization

You learned a few things about initializing weights in Initializing the Weights, on page 137. In case you don't remember that section, here's the one-sentence summary: to avoid squandering a neural network's power, initialize its weights with values that are random and small.

That "random and small" principle, however, doesn't give you concrete numbers. For that, you can use a formula such as *Xavier initialization*, also known as *Glorot initialization*. (Both names come from Xavier Glorot, the dude who proposed it.)

Xavier initialization comes in a few variants. They all give you an approximate range to initialize the weights, based on the number of nodes connected to them. One common variant gives you this this range:

$$|w| \leq \sqrt{\frac{2}{nodes\_in\_layer}}$$

The core concept of Xavier initialization is that the more nodes you have in a layer, the smaller the weights. Intuitively, that means that it doesn't matter how many nodes you have in a layer—the weighted sum of the nodes stays about the same size. Without Xavier initialization, a layer with many nodes would generate a large weighted sum, and that large number could cause problems like dead neurons and vanishing or exploding gradients.

Even though I didn't mention Xavier initialization so far, we already used it: it's the default initializer in Keras. If you want to replace it with another initialization method, of which Keras has a few, use the `kernel_initializer` argument. For example, here is a layer that uses an alternate weight initialization method called *He normal*:

```
model.add(Dense(100, kernel_initializer='he_normal'))
```

Gradient Descent on Steroids

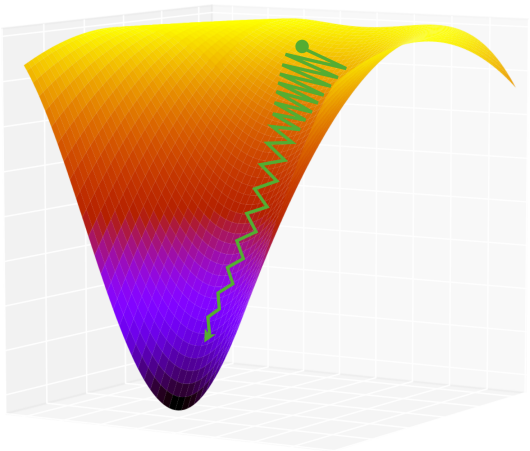
If something stayed unchanged through this book, it's the gradient descent algorithm. We changed the way we compute that gradient, from simple derivatives to backpropagation, but so far, the "descent" part is the same as I introduced it in the first chapters: multiply the gradient by the learning rate and take a step in the opposite direction.

Modern GD, however, can be subtler than that. In Keras, you can pass additional parameters to the `SGD` algorithm:

```
model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1, decay=1e-6, momentum=0.9),
              metrics=['accuracy'])
```

This code includes two new hyperparameters that tweak `SGD`. To understand `decay`, remember that the learning rate is a trade-off: the smaller it is, the smaller each steps of GD—that makes the algorithm more precise, but also slower. When you use `decay`, the learning rate decreases a bit at each step. A well-configured decay causes GD to take big leaps in the beginning of training, when you usually need speed, and baby steps near the end, when you'd rather have precision. This twist on GD is called *learning rate decay*, that's a refreshingly descriptive name.

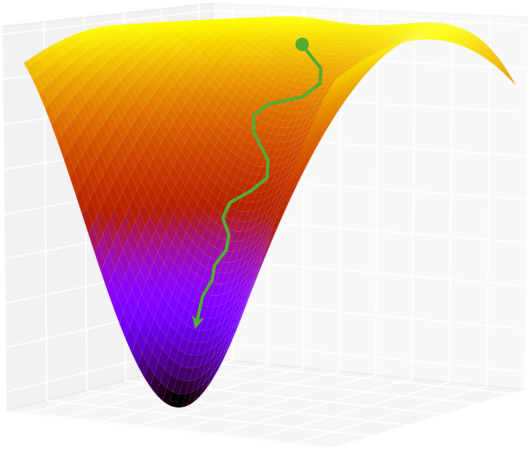
The *momentum* hyperparameter is even subtler. When I introduced GD, you learned that this algorithm has trouble with certain surfaces. For example, it might get stuck into local minima—that is, "holes" in the loss. Another troublesome situation can happen around "canyons" like this one:



GD always moves downhill in the direction of the steeper gradient. In the upper part of this surface, the walls of the canyon are steeper than the path toward the minimum—so GD ends up bouncing back and forth between those walls, barely moving toward the minimum at all.

For this example, I drew a hypothetical path on a three-dimensional surface. However, cases such as this one are common in real life: higher-dimensional losses. When they happen, the loss might stop decreasing for many epochs in a row, leading you to believe that GD has reached a minimum, and giving up on training.

That's where the momentum algorithm enters the scene. Momentum counters the situation I just described by adding an "acceleration" component to GD. That makes for a smoother, less jagged path:



Momentum can speed up training tremendously. In some cases, it may even help GD zip over local minima, propelling it toward the lowest loss. The result is not only faster training, but also higher accuracy.

In Keras, decay and momentum are additional parameters to the standard `SGD` algorithm—but Keras also comes with other implementations of GD, that it calls "optimizers." Instead of `SGD`, you can use an algorithm called `RMSprop` that implements a concept similar to momentum. I sneakily used `RMSprop` in our first deep network in Chapter 16, A Deeper Kind of Network, on page 195. It made the network's training radically faster and more efficient than `SGD`:

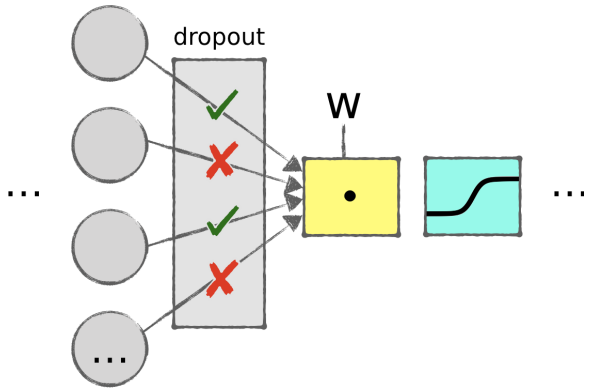
```
model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(lr=0.001),
              metrics=['accuracy'])
```

Finally, take a look at the `Adam` optimizer. It merges momentum and `RMSprop` into one mean algorithm. No wonder it's so popular these days.

Advanced Regularization

When it comes to overfitting, deep neural networks need all the help they can get. In the previous chapter you learned about the classic regularization techniques based on weight decay. However, more modern techniques often work better. One in particular, called *dropout*, is very effective—and also somewhat weird.

Dropout is based on a striking premise: you can reduce overfitting by randomly turning off some nodes in the network. You can see dropout as a filter attached to a layer that randomly disconnects some nodes during each iteration:



Disconnected nodes don't impact the next layers, and they're ignored by backpropagation. It's like they cease to exist until the next iteration.

To use dropout in Keras, you add `Dropout` layers on top of regular hidden layers. You can specify the ratio of nodes to turn off at each iteration—in this example, 25%:

```
from keras.layers import Dropout

model = Sequential()
model.add(Dense(500, activation='sigmoid'))
model.add(Dropout(0.25))
model.add(Dense(200, activation='sigmoid'))
model.add(Dropout(0.25))
model.add(Dense(10, activation='softmax'))
```

What's a Layer?

As it turns out, the notion of a "layer" in neural networks depends on whom you ask. Look at the Keras network in Advanced Regularization, on page 240. In our convention, that network has four layers: the implicit input layer, two hidden layers that use dropout, and an output layer. Keras, on the other hand, calls "layer" everything that you `add()` to the model, `Dropouts` included.

That inconsistency is unfortunate—but it's usually clear from the context whether we're talking about an architectural layer with nodes and weights, or a physical layer such as a `Dropout`.

I've just described how dropout works—but not *why* it works. It's kinda hard to understand intuitively why dropout reduces overfitting, but here's a shot

at it: dropout forces the network to learn in a slightly different way at each iteration of training. In a sense, dropout reshapes one big network into many smaller networks, each of which might learn a different facet of the data. Where a big network is prone to memorize the training set, each small network ends up learning the data its own way, and their combined knowledge is less likely to overfit the data.

That's only one of a few possible ways to explain the effect of dropout. Whatever our intuitive understanding, however, dropout works—and that's what counts. It's one of the first regularization techniques I reach for in the presence of overfitting.

Speaking about things that work, although it's hard to see why: there is one last technique I want to tell you about in this chapter.

One Last Trick: Batch Normalization

Think back to the idea of standardizing data, which we looked at in Standardizing Input Variables, on page 177. In a sentence, a standardized dataset is centered on zero and doesn't stray far from there. As I explained when I introduced this topic, neural networks like data with that shape.

Standardization, however, can only go so far. The data changes as it moves through the layers, losing its network-friendly shape. It would be nice if each hidden layer received standardized inputs. That's pretty much what *batch normalization* does: it re-standardizes data before each layer, for each batch that traverses the network.

Batch normalization involves a few technical complexities. For one, it doesn't necessarily use an average of 0 and a standard deviation of 1, like regular input standardization. Instead, its average and standard deviation are themselves learnable parameters that are tuned by gradient descent. However, you can leave those technical details to Keras and use batch normalization as a black box:

```
from keras.layers import BatchNormalization

model = Sequential()
model.add(Dense(500, activation='sigmoid'))
model.add(BatchNormalization())
model.add(Dense(200, activation='sigmoid'))
model.add(BatchNormalization())
model.add(Dense(10, activation='softmax'))
```

When it was introduced (around 2015), batch normalization was hailed as a breakthrough. You might say that it's an advanced technique—but even

beginners are keen to use it because it works so darn well. It often improves a network's accuracy, and sometimes even speeds up training and reduces overfitting. There is no such thing as an easy win in deep learning, but batch normalization comes as close as anything.

What You Just Learned

Deep neural networks can be wild, untrainable beasts. In this chapter, you learned of a few useful techniques to turn them into cute docile puppies.

We started with a lengthy discussion of *activation functions*. You learned that nonlinear functions are a necessity in neural networks, but they must be chosen carefully. The sigmoid, that we used so far, can cause a number of problems in deep networks: saddening *dead neurons*, perplexing *vanishing gradients*, and shocking *exploding gradients*. For that reason, we looked at a few alternatives to the sigmoid—in particular, the popular *ReLU* activation function.

To round up this chapter, we made a whirlwind tour through a number of other techniques that help us tame deep neural networks:

- *Xavier initialization* to initialize a neural network's weights.
- A handful of advanced GD algorithms: *learning rate decay*, *momentum*, *RMSprop* and *Adam*.
- *Dropout*, a brilliantly counterintuitive regularization technique.
- The extremely useful technique of *batch normalization*.

You're probably itching to try out these techniques yourself. If so, then the next hands-on exercise will scratch that itch. Otherwise, prepare for the next chapter: it's going to take everything that you learned so far about neural networks and spin it on its head.

Hands On: The 10 Epochs Challenge

Now that you have a few tricks under your belt, you can start experimenting with deep neural networks on your own. Speaking of which, I have a challenge for you.

To set up this challenge, I wrote a neural network and trained it on the MNIST data set. You already saw plenty of MNIST classifiers, but this time, it's a deep neural network written in Keras. Go and take a look at it. It's in `network_mnist.py`, in this chapter's source code.

This network doesn't use any of the techniques from this chapter, but it still does okay. I trained it for 10 epochs and got this result:

```
loss: 0.2026 - acc: 0.9388 - val_loss: 0.2487 - val_acc: 0.9228
```

Now it's your turn. You mission, should you choose to accept it, is to improve this network's accuracy while respecting two constraints:

1. Don't change the network's layout. Keep the same number of layers and nodes per layers.
2. Don't change the number of epochs or the batch size. Keep them at 10 and 32, respectively.

In all other respects, you can go wild. You can use the tricks that we described in this chapter, or even explore new ones—activation functions, optimization algorithms, or whatever else you learn by browsing Keras's documentation.[1]

Here are a few ideas to get you up and running:

- Replace the sigmoids with ReLUs or Leaky ReLUs.
- Switch from `SGD` to another GD algorithm.
- Add dropouts.
- Add batch normalization.

If your network does better than 97.5% on the validation set, then victory is yours.

Although I'm pretty sure that you can hit that 97.5%, I'd be surprised if you did much better than that. The techniques that we introduced in this chapter are essential, but they can only take us so far. To get radically better results, we need something more than incremental tricks—we need a fundamentally different neural network architecture. That is the subject of the next chapter.

1. keras.io

CHAPTER 19

An Adventure in Convolution

Content to be supplied later.

CHAPTER 20

Understanding Deep Learning

Content to be supplied later.

CHAPTER 21

This Is Only the Beginning

Content to be supplied later.

APPENDIX 1

Just Enough Python

Welcome to Python! These days, Python is *the* language for machine learning. From the research labs to the biggest AI companies, Python snaked its way everywhere. (See what I did here?)

Python is a great language all around, but it fits machine learning particularly well—for two reasons. First, it comes with powerful numerical and scientific libraries. Second, Python is powerful, but also easy to approach. Many people in the ML community lack a background in programming, and would be intimidated by a less friendly language.

Don't get me wrong: Python isn't a toy language that you can learn in a day. It comes with sophisticated features that take time to learn and master. The examples in this book, however, avoid those features in favour of a small subset of Python. These pages are a day trip through that mini-Python. Read them, and you'll know just enough of the language to read and modify the code in this book.

Here is what we'll talk about:

- We'll see what Python code looks like, and describe the fundamental features of the language.
- We'll get a whirldwind tour of Python's basic building blocks: types such as strings and collections, and control structures such as loops.
- We'll learn to define and call Python functions.
- Finally, we'll learn how to work with multiple files and how to install libraries.

At the end, I'll tell you where to look for more in-depth information.

This appendix moves at a much faster pace than the rest of the book. We won't linger on basic programming concepts that you already know. In most cases, we'll glance at a few lines of code, see what Python's syntax looks like,

and move forward. Maybe have a browser handy as you read through, in case you want to dig a bit deeper into a specific Python feature.

While you're at your computer, you might want to type Python statements as you read through. Open a terminal and check that you have Python 3 installed.

```
python3 --version
```

If the `python3` command fails, then try the `python` command, without a version number—but make sure that your version of Python is 3.0 or later. If you don't have Python, or you have an earlier version, then look on the Internet[1] for an installation method that works for you and your system.

There are two main ways to run Python code. One is to write a Python program in a file with a `.py` extension, and execute it:

```
python3 my_code.py
```

You can also execute Python code in an interactive Python interpreter. Start it with the `python3` command, without a file name. Once the interpreter is running, you can execute Python statements on the fly—like this one:

```
print("Hello, Python!")
```

Oh, by the way: it's always embarrassing when you don't know how to close an interactive interpreter. In Python, you do that with the `exit()` command.

Jupyter Notebooks

Being a developer, you're used to code with IDEs, or maybe a combo of a text editor and the command line. By contrast, most people in the ML community write their code in a tool called Jupyter Notebook[a]. Jupyter allows you to type and run Python code in a web page, mixed with descriptive text. Imagine a simple in-browser word processor that also executes code snippets, and you pretty much get the idea.

Jupyter has two advantages over the classic developer's toolset:

1. It allows you to mix Python code with formatted text, images, and data visualizations like charts and tables. That's a big deal for data scientists, who often need to explain the context, input, and output of their system.

2. With their gigantic toolbars and hundreds of keyboard shortcuts, IDEs are intimidating. As for text editors like vim… if you're into those, then you know that they can positively scare off a beginner. By contrast, Jupyter is friendly and welcoming, especially to people who lack a programming background. Jupyter allows them to type and run code in the familiar context of a web page.

1. realpython.com/installing-python

You don't have to use Jupyter to read this book. All the examples in these pages come as *.py files, and you should feel free to edit them in your favourite IDE or text editor. If you already know and like Jupyter, however, you'll also find notebook versions of all the examples in the notebooks directory.

a. jupyter.org

What Python Looks Like

Python code is famously easy to read. It has even been described half-jokingly as "executable pseudocode". For example, you can probably understand a lot of the following program, even if you have no Python experience at all:

just_enough_python/find_prime_numbers.py

```
# Import the square root function
from math import sqrt

# Returns True if the argument is a prime number, False otherwise
def is_prime(n):
    # n is prime if it cannot be divided evenly by any number in
    # the range from 2 to the square root of n.
    max_check_value = sqrt(n)
    # range(a, b) goes from a included to b excluded. Both a and b
    # must be integers, so we convert max_check_value to an integer
    # with the in-built int() function.
    for x in range(2, int(max_check_value) + 1):
        # Check the remainder of the integer division of n by x
        if n % x == 0:
            return False  # n can be divided by x, so it's not prime
    return True  # n is prime

MAX_RANGE = 100
primes = []
print("Computing the prime numbers from 2 to %d:" % MAX_RANGE)
for n in range(2, MAX_RANGE):
    if is_prime(n):
        primes.append(n)
print(primes)
```

The program above finds the prime numbers in the range from 2 to 100. It's short, but it uses most of the features of Python that you need for this book. If you can understand all of it, then you shouldn't have trouble understanding the examples from the previous chapters. If you can't… well, that's what this appendix is all about.

A lot of Python's readability is a consequence of two features of the language: dynamic typing, and significant whitespace. Let's discuss them.

Python Is Dynamically Typed

In Python, you don't declare the types of variables. You just assign a value to them, and they spring into existence:

```
x = "Hello!"
x = 42
```

In the code above, the variable x doesn't have a type constraint, so it can hold values of any type—in the two lines above, a string and an integer.

As a programmer, you probably have a strong opinion about type systems. If you're used to languages such as JavaScript or Ruby, then you might appreciate the terseness and versatility of dynamic typing. On the other hand, if your daily languages is Java, C#, or C, then you might favor the safety and consistency of explicit type declarations, such as `int x = 42`.

If you're not used to dynamic typing, I suggest you give it a fair chance. It's always worth having a dynamically typed language like Python in your toolbox.

In Python, Indentation Matters

Python has a unique way to delimit a block of code, like the body of a function, or the body of a loop. For that purpose, most languages use special characters such as curly braces, or keywords like `begin` and `end`. By contrast, you delimit Python blocks by indenting them. Look at this contrived snippet of code:

```
x = 7
if x > 5:
    print("x is greater than 5")
    print("And while we're here...")
    if x < 10:
        print("...it's also smaller than 10")
else:
    print("x is less or equal than 5")
```

The code between the `if x > 5` statement and the `else` statement is a single block, because it's all indented the same. The line after the `if x < 10` statement is a nested one-line block, because it's indented further. The line after the `else` is also a one-line block. The program above prints:

```
x is greater than 5
And while we're here...
...it's also smaller than 10
```

You can indent the lines in a block with either spaces or tabs, as long as you do so consistently. Whitespace actually has a meaning in Python!

Significant whitespace is by far the most controversial feature of the language. Some programmers love it. For others… well, it can be an acquired taste. If you're in the second camp, consider the advantages of significant whitespace: it keeps the indentation style consistent across different programmers, and it removes the need for noisy braces and semicolons.

Readable by Convention

Look at Python code from multiple programmers, and you'll see that it tends to have a common, uniform style. We discuss one reason for that consistency in the section In Python, Indentation Matters, on page 254. Another reason is that the Python community is serious about coding conventions. They even have an official stylesheet that is revered by coders and automatically checked by IDEs.[a]

Here are a couple of examples of consistent style in Python. Almost all programmers use "snake case" for the names of Python's variables and functions, as in `this_is_a_variable`, or `is_prime()`. Python has no language-level support for constants, but if you want to let the reader know that a variable is meant to be a constant, then you name it with uppercase letters separated by underscores, as in `MAX_RANGE`.

Fair warning: ignoring these naming conventions will expose you to frowning by vocal pythonistas.

a. www.python.org/dev/peps/pep-0008/

Now that you know what Python code looks like, let's get into a bit more detail.

Python's Building Blocks

I don't know about you—for me, the boring part of learning a new language is the mandatory listing of basic types, operators, and control structures. Whenever I try to memorize the finer points of loop syntax and operator precedence, that information slips right off my brain. It only sticks once I start writing code.

For that reason, I won't bog you down in details that you can look up yourself later, when you're confronted with real code. I'll keep this section on types, operators and control structures as quick as I can. We'll have a broad overview, only zooming in on those few details that are important for this book.

Data Types and Operators

Python comes with the data types you expect: integers, floating point numbers, booleans, strings—the works:

```
an_integer = 42
a_float = 0.5
a_boolean = True
a_string = "abc"
```

As for Python's operators, you'll probably find them familiar across the board:

```
an_integer + a_float   # => 42.5
an_integer >= a_float  # => True
a_float * 2            # => 1.0
an_integer / 10        # => 4.2
an_integer % 10        # => 2
not a_boolean          # => False
a_boolean or False     # => True
a_string + 'def'       # => 'abcdef'
```

Compared to some other dynamic languages like JavaScript or Perl, Python is strict about mixing types. While the language does cast types automatically in some cases, potentially ambigous operations usually result in an error:

```
# Implicit cast from an integer to a boolean:
not 10     # => False

# Failed implicit cast from an integer to a string:
"20" + 20  # => TypeError: must be str, not int
```

If you need to convert Python values from one type to the other, you generally do so explicitly, with functions such as str(), int() and bool():

```
"20" + str(20)  # => '2020'
```

We'll get a closer look at Python's strings in a minute—but first, let's look at variables that contain multiple values.

Collections

Every modern language supports variables with multiple ordered values, usually called "arrays", "lists", or "vectors". Python has two such multi-valued collection types: *lists* and *tuples*.

Here is what tuples look like:

just_enough_python/collections.py

```
a_tuple = (3, 9, 12, 7, 1, -4)
len(a_tuple)  # => 6
a_tuple[2]    # => 12
a_tuple[2:5]  # => (12, 7, 1)
```

You create a tuple by wrapping a sequence of comma-separated values in round brackets. The code above demonstrates a couple of things you can do with tuples: getting their length, and getting one or more of their elements (using zero-based indexes).

Tuples are immutable: once you create one, it cannot change. If you want a mutable collection of values… that's what lists are for. They look similar to tuples—only they use square brackets, and they are mutable:

```
a_list = [10, 20, 30]
a_list[1] = a_list[1] + 2
a_list.append(100)
a_list  # => [10, 22, 30, 100]
```

Python being dynamic, tuples and lists can contain values of any type, or even a mix of types. As an example, each element in the following list has a different type, including a tuple and another list:

```
a_mixed_list = ['a', 42, False, (10, 20), 99.9, a_list]
```

Lists and tuples will both appear in this book's source code—but they won't get the spotlight. The most common collection we'll use, by far, is the powerful `array` type from the NumPy library. There's no need to linger on arrays here, because you're going to get familiar with them as you read the book. Just know that they look a bit like Python's lists, but they have multiple dimensions—so they're perfect to represent matrices, that are a common data structure in machine learning.

In this appendix, we're going to skip over the *dictionary*—Python's take on key-value collections. The reason we're passing over dictionaries is that we don't happen to use them in the book. However, if you want to do some coding of your own, including ML code, then dictionaries are one of the first features you should look up.

We're almost done with Python's basic types. We only need to give a closer look at strings.

Strings

You can define a string with either double or single quotes:

just_enough_python/strings.py

```
s1 = "This is a string"
s2 = 'this is also a string'
s1 + " and " + s2  # => 'This is a string and this is also a string'
```

Python programmers tend to use double quotes, and switch to single quotes when the string itself contains double quotes:

```
print('Yup, this is yet another "Hello, World!" example')
```

A string behaves pretty much like a tuple of characters, which is why we looked at collections first. In particular, you can index a string's individual characters:

```
s3 = s2[8:12]
s3  # => 'also'
```

There is another way in which Python's strings look like tuples, and it will please the functional programmers amongst you (you know who you are): strings are immutable. You cannot modify a string in place—you have to create a new string, as I did in the code above.

One feature that we use a lot in this book is string interpolation—a way to embed variables in a string. Python offers a few ways to do string interpolation, but we're going to stick with the "classic" style, because it's compatible with all modern versions of Python. It looks like this:

```
a = 1
b = 99.12345
c = 'X'
"The values of these variables are %d, %.2f, and %s" % (a, b, c)
# => The values of these variables are 1, 99.12, and X
```

The % sign separates the string from a tuple of values that must be embedded in it. The embedding positions are marked by special codes, that are also prefixed by a %. In particular, %d means "embed as a decimal number", %s means "embed as a string", and %.2f means "embed as a floating point number rounded to two decimal places".

Finally, if you want to print a percent sign inside an interpolated string, you need to escape it by typing a double percent signs:

```
"Less than %.d%% of ML books have a hammer on the cover" % a
# => Less than 1% of ML books have a hammer on the cover
```

Loops

Python comes with the usual C-style control structures: if, while, for, and their ilk. We already saw how to use if in our first Python example, in What Python Looks Like, on page 253. As for while, we won't use it in this book.

On the other hand, we'll use for loops all the time. Here is what they look like:

just_enough_python/loop.py
```
for i in range(4):
    if i % 2 == 0:
        print("%d is an even number" % i)
    else:
        print("%d is an odd number" % i)
```

The for loop above iterates over the values in the range, from 0 to 4 included:

```
0 is an even number
1 is an odd number
2 is an even number
3 is an odd number
```

Note that experienced Python coders tend to shun for loops, in favor of more elegant constructs inspired by functional programming. If you're curious, check out Python's list comprehension.[2]

With that, we've completed our tour of Python's basic types, operators and control structures. Let's move on to another fundamental construct of the language: functions.

Defining and Calling Functions

In most programming languages, you cannot go far without defining and calling your own functions. In Python, you define a function with the def keyword:

just_enough_python/functions.py
```
def welcome(user):
    PASSWORD = "1234"
    message = "Hi, %s! Your new password is %s" % (user, PASSWORD)
    return message
```

The function above generates a password for a new user. (Admittedly, the default password isn't the most secure—but hey, at least it's popular.) Then it composes a welcome message, and returns the message to the caller.

Pythonically, you don't need to delimit the function's body with brackets—you just indent it. Also, being Python dynamically typed, you don't need to specify the type of the user parameter, or the type of the function's return value. Taken together, these features make for a very concise function declaration.

Once you have a function defined, you can call it:

```
welcome("Roberto")  # => 'Hi, Roberto! Your new password is 1234'
```

2. docs.python.org/3/tutorial/datastructures.html

Changing Your Arguments

In case you're wondering, Python's function parameters are passed by reference, not by copy. That means that if you pass a mutable argument to a function, the function might modify it. Here is a function that adds an element to a list:

```
def modify_list(l):
    l.append(42)

a_list = [1, 2, 3]
modify_list(a_list)
a_list  # => [1, 2, 3, 42]
```

In most cases, you don't want to have your arguments modified behind your back. Polite functions generally don't modify their arguments, and treat them as if they were read-only—unless modifying the argument is the entire point of the function, as in the case above.

Python has a few different flavors of function arguments. Let's look at them.

Named Arguments

You can use *named arguments*, also called *keyword arguments*, to make a function call easier to read. Let's look at an example.

Assume that your boss just saw the welcome() function from the previous section, and he really liked it—but he's concerned that the password isn't very secure. He proposes a solution: add a secure argument to the function. If secure is True, then the function should generate a more secure password.

Here is the updated welcome() function:

```
def welcome(user, secure):
    if secure:
        PASSWORD = "123456"
    else:
        PASSWORD = "1234"
    return "Hi, %s! Your new password is %s" % (user, PASSWORD)
```

We don't mess around with security here, buddy. Now we can call the function with the secure flag on:

```
welcome("Roberto", True)  # => Hi, Roberto! Your new password is 123456
```

Note, however, that the function call above has a readability issue: if you don't know what that True argument means, then you have no way of finding out, short of looking at the function definition. It would be nice to show clearly that True is the value of the secure argument. You can do that by using named arguments:

```
welcome("Roberto", secure=True)  # => Hi, Roberto! Your new password is 123456
```

Now the call is more readable. As an added bonus, we can change the order of the arguments in the call—although I'll leave it to you to decide whether that's a good idea in general:

```
welcome(secure=False, user="Mike")  # => Hi, Mike! Your new password is 1234
```

We're not quite done with function arguments yet. There is one last useful feature related to them, and we use it a lot in this book.

Default Arguments

The boss just asked for a few more changes to the security system. He wants the function to work even if we don't provide the user's name. Besides, he wants the password to be secure by default, unless the caller specificies otherwise.

We can implement both features by specifying the arguments' default values in the function definition:

```
def welcome(user="dear user", secure=True):
    if secure:
        PASSWORD = "123456"
    else:
        PASSWORD = "1234"
    return "Hi, %s! Your new password is %s" % (user, PASSWORD)
```

Now we can skip one or both arguments, and they'll take their default values:

```
welcome()  # => Hi, dear user! Your new password is 123456
```

You can even mix and match named arguments and default arguments:

```
welcome(secure=False)  # => Hi, dear user! Your new password is 1234
```

With that, you know everything you need about Python functions and their arguments. And just to be clear: I do not recommend the code above to generate passwords in production. You never know… it might contain non-obvious security bugs.

Working With Modules and Packages

A short Python program can happily live in a single file—but as soon as you write a larger program, you need a way to organize its code. Above functions, Python has two more levels of code organization: functions and other code live in *modules*, and modules live in *packages*. Let's start by looking at modules.

Defining and Importing Modules

A module defines entities such as constants and functions, that you can import and use in a program. Aside from some of the in-built modules of the Python interpreter, a module is a Python file.

For example, here is a module named my_module.py:

just_enough_python/my_module.py
```
THE_ANSWER = 42

def ask():
    return THE_ANSWER
```

The file above defines a function and a constant. Now imagine that we have a Python program in the same directory. This program can import either (or both) definitions with the import keyword:

just_enough_python/my_program.py
```
from my_module import ask, THE_ANSWER
```

When you import a module, two things happen:

1. the code in the module is executed;
2. the names you imported become available in your program.

For example, now my_program can call the ask() function:

```
ask()        # => 42
```

Note that the code in the module is executed only the first time you import it. If you import a module more than once, Python marks it as "already imported" the first time around, and ignores subsequent imports.

Instead of cherry-picking the names you want to import, like in the example above, you can also import the entire module:

just_enough_python/import_everything.py
```
import my_module
```

The line above will import all the names defined in my_module.py. When you import an entire module like that, however, those names might clash with other names in the main program, or in another module. To avoid those clashes, Python forces you to prefix the names with the name of the module, like this:

```
my_module.ask()        # => 42
my_module.THE_ANSWER   # => 42
```

To avoid prefixing the same long module name dozens of time, you can give a shorter name to the module when you import it, like this:

just_enough_python/module_renaming.py
```
import my_module as mm
mm.ask()  # => 42
```

For example, the numpy library is almost always shortened to np, like this:

```
import numpy as np
```

After the renaming, you can reference to the functions in this library with the np prefix, as in np.multiply(x, y).

The Standard Library

Python prides itself on being "batteries included", meaning that it comes with a bunch of useful modules right out of the box. For example, the math module gives you the basic mathematical functions and constants:

```
import math
math.sqrt(16)  # => 4.0
math.pi        # => 3.141592653589793
```

Note that math.pi doesn't follow Python's conventions for the names of constants—if it did, then it would be called math.PI. The standard library is showing its years here, as that name dates back earlier than the current naming conventions.

The main Idiom

When it comes to modules, we have one last topic to mention. It concerns a very common Python idiom.

We said that a file of Python code can be either a program, or a module, depending on how you use it. You execute a program directly, with a command like python3 my_code.py. By contrast, you use a module by importing it in another file.

However, it's common for the same Python file to play both roles. A file with this binary nature can either be run as a stand-alone program, or imported as a module. Here is one such file:

just_enough_python/greetings.py
```
print("Executing the code in greetings.py")

def greet(name):
    print("Hello,", name)

if __name__ == "__main__":
```

```
    greet("human")
```

Ignore the last two lines in the file above for a minute. If we import greetings.py from another file, the usual things happen: first, the code in greetings.py is executed; and second, we can access the greet() function:

just_enough_python/greetings_demo.py

```
import greetings
greetings.greet("Bill")
```

If you run python3 greetings_demo.py, you get:

```
Executing the code in greetings.py
Hello, Bill
```

However, you can also run greetings.py as a stand-alone program, by typing python3 greetings.py. In that case, you get:

```
Executing the code in greetings.py
Hello, human
```

The secret to running the file as a program is in the idiom if __name__ == "__main__". (That's a double underscore both before and after name and main.) This idiom stands for: "only execute the following code if this file is run directly". By contrast, if the file gets imported, then the Python interpreter skips the if block.

To see how this idiom is useful, imagine writing a program that defines a bunch of functions, and then uses those functions to interact with the user. When you load the file from another program, you want to skip the user interaction—but you still want to access the functions, to reuse or test them. You can fence the user interaction behind the if... "__main__" idiom, and it will be executed only when the file runs as the main program.

You'll see the if... "__main__" idiom throughout this book, and in the source code of most Python libraries.

Managing Packages

Above modules, packages are the next level of code organization. A package is essentially a bundle of modules, organized in a directory structure.

In this book, we don't define our own packages—but we use them all the time, for one reason: when you install a Python library, that library comes in the form of a package.

There are multiple ways to install Python libraries. Most Python developers use the pip package manager. Others prefer an alternative tool named Conda. Let's look at both tools.

Installing Packages with pip

pip[3] is Python's official package manager. Its name is a recursive acronym that stands for "pip Installs Packages". (Yup, the Python community has a warped sense of humor. After all, the name of the language is a homage to the Monty Pythons.)

If you have Python installed, chances are you also have pip. You can use it to install one of the many packages in PyPI[4], that stands for "Python Package Index"—Python's official package repository. For example, this command installs version 2.2.4 of the Keras machine learning library:

```
pip install keras==2.2.4
```

Once you have Keras installed, you can use its modules from a Python program. This line imports the serialize() function from the keras.metrics module:

```
from keras.metrics import serialize
```

To be precise, keras is a module in the Keras library, and metrics is a submodule of keras.

pip has all the features you expect in a package manager: you can install a specific version of a package, list the packages installed, and so on. If you're looking for a simple out-of-the-box system to install libraries, pip has you covered. If you want something more sophisticated… then keep reading.

Installing Packages with Conda

Conda[5] is the package manager of choice in the ML community. It's part of a hefty Python distribution called Anaconda[6], that's especially taylored to data science.

Anaconda comes with a lot of bells and whistles, including an IDE and its own repository of packages, separated from the official Python one. If you don't need the extras, then you can install Miniconda[7], which is a much slimmer install that only includes Conda and Python.

3. pip.pypa.io
4. pypi.org
5. conda.io
6. www.anaconda.com
7. docs.conda.io/en/latest/miniconda.html

When it comes to installing a package, Conda works pretty much the same as pip:

```
conda install keras=2.2.4
```

However, Conda has a couple of selling points over pip. For one, where pip focuses on Python libraries, Conda can handle data science packages written in different languages. Also, Conda allows you to create "environments" that you can activate and deactivate on the fly. Each environment can have a different set of libraries. By contrast, packages installed with pip are global: all the Python code on your machine sees the same version of the package.

Conda also integrates well with pip: if you want a package that's only available in the PyPI repository, but not in Conda's repository, you can run `pip install` in a Conda environment, and the package will only be visible in that environment.

To sum it up, the choice between pip and Conda usually boils down to this: if you're OK with globally installed packages, then use pip; if you prefer to maintain separate environments that contain different packages, for example a different set of packages for each project, then use Conda.

You'll need to install a few packages to run the code in this book. Setting Up Your System, on page 11 contains instructions to install them with pip. If you opt for Conda, then take a look at the `readme.txt` in the book's source code.

Creating and Using Objects

Python is an *object-oriented* language. The code in this book, however, doesn't make much use of objects. This short section tells you everything you need to know about them.

You can see a Python *object* as a special kind of variable. Regular variables belong to the language's built-in types, such as `float` or `bool`. By contrast, objects belong to types that are defined by yourself, or by a library. These higher-level types are called *classes*.

For example, Python has no built-in type for dates—but the standard library `datetime` defines a class named `date`. Import that library, and you can create `date` objects:

just_enough_python/objects.py

```
from datetime import date
moon_landing = date(1969, 7, 20)
```

Running the two lines above creates a variable called `moon_landing` that contains a `date`. To create that date, we passed it a year, a month, and a day.

In some other languages, you use the keyword new to create an object. In Python, you just use the name of the class, followed by parentheses. You pass the object creation parameters inside the parentheses, just like you would for a regular function call.

Once you have an object, you can call its *methods*. Methods are like functions that are specific to the object's class. For example, dates have a weekday() method that returns the day of the week, from 0 to 6:

```
moon_landing.weekday()  # => 6
```

Now we know that the moon landing happened on a Sunday.

Just like functions, some methods take arguments. For example, the date class has a replace() method that returns a copy of the date with a different year, month, or day. It also has a format() method that formats the data according to a format string. Here's a snippet of code that uses those two methods:

```
viking_1_mars_landing = moon_landing.replace(year=1976)
viking_1_mars_landing.strftime("%d/%m/%y")  # => 20/07/76
```

Also like functions, methods can have default arguments, and can be called with named arguments.

For historical reasons, the name of the date class is all lowercase—but today it's customary to use camel case for class names, as in ThisIsAClass. For example, the Keras library that we use in Part III of this book defines classes with names like Sequential, BatchNormalization, and MaxPooling2D.

In some languages, such Java and C#, you cannot even write a basic program without classes and objects. By contrast, Python's object-oriented features are almost optional. You can easily write a procedural Python program without any objects or classes… Or at least, that's until you take a deeper look, and you find out that objects and classes are woven right into Python's fabric. For example, strings are actually objects with their own methods:

```
'strings are objects'.upper()  # => 'STRINGS ARE OBJECTS'
```

Even if objects and classes are core to Python, however, you can go pretty far without using them much. In particular, we use objects in a very limited fashion in this book. We don't define our own classes—we just import classes from libraries such as Keras. We use those classes to create objects, and we call those objects' methods. That's pretty much all the object-oriented programming you need here.

That's It, Folks!

Congratulations for reading this far! Now you know enough Python to hack the code in this book.

Admittedly, we've only scratched the surface of this beautiful programming language. We didn't look at the more advanced and elegant Python constructs. We didn't even mention some of the features that you need to write professional software, such as error management. However, those features aren't necessary to follow the examples in this book. You can study them later, once you decide to get deeper into Python.

Becoming a Python Expert

Your first experiences with Python might whet your appetite. If you decide to learn more about the language, then you have only one problem: too many choices. Python is one of the most popular learning topics around. You can find Python courses, primers, and tutorials all around the Internet, and in your local bookshop. The popular online training hubs, such as Udemy, Coursera, Udacity, and Pluralsight, all have their own Python online courses—sometimes many of them, often available for free.

With so many options, your best bet is to Google for something that fits your needs. If you prefer to browse through a list of resources, however, you can find one on the official Python site.[a] The same site carries a list of introductory books[b], and even an official tutorial.[c]

---

a. wiki.python.org/moin/BeginnersGuide/Programmers
b. wiki.python.org/moin/IntroductoryBooks
c. docs.python.org/3/tutorial

Enjoy the book, and enjoy Python!

APPENDIX 2

The Words of Machine Learning

When I took my first steps into machine learning, one of the biggest hurdles for me was learning the vocabulary. So many words in machine learning sound familiar, and yet subtly foreign. Am I supposed to already know what a "feature" is? What does "numerically stable" mean, exactly?

This appendix gives you quick definitions of common terms and expressions of machine learning. Most entries also refer you to the section in the book where a term is first introduced, in case you need something more than a quick reminder. These definitions have no pretense to be Wikipedia-worthy. They're here just to trigger your memory and point you at additional information.

You won't find *all* the terms from the book in here—only those that I thought you might want to review. If you can't find the word you're looking for, then look it up in the book's index.

In the definitions, terms written in *italic* have their own entry in this appendix.

Activation function

The function that follows the *weighted sum* in each *layer* of a *neural network*—in this book, typically a *sigmoid*, a *softmax*, or a *ReLU*. I introduce the name "activation function" to refer to the sigmoid of the *perceptron* in Chapter 8, The Perceptron, on page 95.

Adam

A variant of *gradient descent*. See Gradient Descent on Steroids, on page 238.

Artificial neural network

A more technically precise name for a *neural network*. There was a time where people felt the need to specify that they're talking about an "artificial" neural network, as opposed to a biological one.

Asymptotic

A curve is asymptotic when it approaches a value without ever quite reaching it. If I say, for example, "this *loss* asymptotically approaches a value of 10," that means that the loss gets closer and closer to 10, but it never quite reaches it.

Backpropagation

The fundamental algorithm to train *neural networks*. Backpropagation—or "backprop," for friends,—calculates the gradients of the *loss* with respect to the *weights*. Chapter 11, Training the Network, on page 127 is dedicated to backpropagation.

Batch gradient descent

The "plain vanilla" version of *gradient descent*, where the gradient is calculated based on all the training examples, taken together as a single batch. By contrast, see *mini-batch gradient descent* and *stochastic gradient descent*.

Bias

In many machine learning systems, including the ones in this book, the bias is one of the learnable *parameters* of the *model*. To see its mathematical meaning, check out Adding a Bias, on page 27 and Adding More Dimensions, on page 48. However, the term "bias" has another meaning in machine learning: you can say that a system has "high bias" when it *underfits* the training data, as I explain in Bias and Variance, on page 215.

Bias-variance tradeoff

Bias and *variance* are alternate terms for *underfitting* and *overfitting*. The "tradeoff" between them is a pragmatic concern: in many *supervised learning* systems, reducing overfitting can cause the system to underfit the training data, and vice versa. This balancing act is explored in Chapter 17, Defeating Overfitting, on page 209.

Bias column

In some machine learning models, the *bias* is a special case amongst *parameters*: while other parameters are used in the *weighted sum*, the bias stands aside. In Bye Bye, Bias, on page 60, I introduce a trick to get rid of this special case: by adding a column full of 1s to the inputs, we turn the bias into a *weight* like any other. This ad hoc column is called the "bias column."

Binary classifier

A *classifier* with an boolean output—for example, one that classifies the state of a hardware component as either "working" or "broken." If the classifier has more than two classes, then it's a *multinomial classifier*.

Broadcasting

A feature of some numerical libraries such as NumPy. Broadcasting allows you to treat arrays like you would treat individual values. For example, you can subtract two NumPy arrays: each element in the result is the subtraction of the matching elements in the two original arrays. You can see examples of broadcasting throughout this book, starting with Implementing Prediction, on page 21 and Upgrading the Loss, on page 57.

Chain rule

The chain rule is the foundation of the *backpropagation* algorithm. It says that the derivative of a composite function is the product of the derivative of the component functions. See The Chain Rule, on page 129.

Classes

The word "class" has different meanings in machine learning and programming. In machine learning, the classes are the possible labels of a classifier. For example, a classifier that recognizes digits has 10 classes, one for each digit; a classifier that recognizes dog breeds has one class per breed; and so on. By contrast, a class in programming is something else entirely—see Creating and Using Objects, on page 266.

Classifier

A system that assigns data to one of a limited set of *classes*. For example, you might label a movie review as "positive," "negative," or "neutral." The machine that rads the review and applies that label is a classifier. I introduce the concept in Chapter 5, A Discerning Machine, on page 65.

Computer vision

The field that studies how computers can recognize images and videos. Today, most computer vision problems are solved with machine learning.

Convergence

In general, an algorithm converges when it finds a result, as opposed to running forever. In machine learning, this word is usually associated with algorithms like *gradient descent*. If GD converges, this means that it manages to minimize the *loss*; if it doesn't, that means that it oscillates indefinitely around the minimum loss, or even that it steps further and further from the minimum.

Convex

A mathematical function is convex when you can pick any two points on it, join them with a line, and the line won't cross any other point in the function. Intuitively, that means that the function has a single "cavity" (like, say, the exponential function) as opposed to multiple ones (like the sinus function).

Cross entropy loss

One of the formulae that we use to calculate the *loss*. It's a generalization of the *log loss*, and it's used for *multinomial classification* problems. See Writing the Classification Functions, on page 121.

Data flow diagram

A diagram that visualizes how data moves through a system. For an example, look at one of the many diagrams in Chapter 11, Training the Network, on page 127.

Dead neuron

The phenomenon of dead neurons happens when an *activation function* in a neural network gets *saturated*, and its *gradient* gets close to 0. When that happens, the *gradient descent* algorithm gets stuck, and the *nodes* that are downhill of the activation function stop learning. We describe this phenomenon in Dead Neurons, on page 139 and The Sigmoid and Its Consequences, on page 229.

Decision boundary

A *classifier* does its job by partitioning the space of the *examples* into areas. The decision boundary is the border between those areas. It's called "decision boundary" because data points on different sides of the boundaries are assigned to different *classes*. I know, that's a pretty abstract definition—but you can find concrete examples in Tracing a Boundary, on page 147.

Deep learning

To put it simply, "deep learning" is a short name for "machine learning done with *neural networks* that have many *layers*." More broadly, the term indicates a large set of techniques that are used today in machine learning, such as *convolutions*, *recurrent neural networks*, and many more.

Densely connected

In a *neural network*, a *layer* is densely connected when each *node* is connected to all the nodes in the previous layer.

Dev set

Another name for the *validation set*.

Derivative

See the definition for *gradient*, and also the one for *partial derivative*.

Differentiable

A function is differentiable when you can calculate its *derivative* at any point. Intuitively, that happens when the function is smooth, and it doesn't have sudden jumps or holes. I use this term in When Gradient Descent Fails, on page 44.

Dot product

A way to multiply two arrays of numbers so that the result is a single number. For the details, see Multiplying Matrices, on page 51.

Dropout

A bizarre (but powerful) *regularization* technique that randomly turns off *nodes* in a training *neural network*. See Advanced Regularization, on page 240.

Early stopping

A technique to counter *overfitting*, described in A Regularization Toolbox, on page 222. Early stopping is about finding the best moment to stop the *training phase*—after the system has learned enough to be useful, but before it starts overfitting the *training set*.

Epoch

In *mini-batch gradient descent* (or *stochastic gradient descent*), a training iteration processes only a fraction of the entire training set. By contrast, an "epoch" is a pass through the entire training set. For example, if you have 10000 training examples, and you process them in batches of 10, it will take 1000 iterations to complete an epoch.

Examples

See *supervised learning*.

Exploding gradient

A counterpart to the *vanishing gradient* problem. In the case of the exploding gradient, the *gradient* grows as it *backpropagates* through a *neural network*. See Don't Play with Explosives, on page 233.

Feature

A common name to indicate the *input variables* in a machine learning system. For example, the pizza forecasting problem from Adding More

Dimensions, on page 48 has three features: "Reservations," "Temperature," and "Tourists." In an image recognition problem, such as the one in Chapter 7, The Final Challenge, on page 85, you can treat each pixel in the input images as a separate feature.

Feature scaling

Machine learning systems tend to work better if their input *features* span a similar range—for example, from 0 to 1. On the other hand, if some features can have much larger values than others, then you might want to rescale them to the same range. That's what "feature scaling" means. Read Preparing Data, on page 176 for more details, or check out the entry on *standardization* for an broader data preparation technique.

Forward propagation

The process by which data moves from the input to the output of a supervised learning system. I use this term for the first time in Confidence and Doubt, on page 68—but it really starts making sense when I introduce *neural networks*, because it parallels the process of *backpropagation*.

Global minimum

The minimum value over an entire function. For example, imagine a function that takes a date and returns the temperature in Chicago on that day. The global minimum of that function is on January 20, 1985, when the thermometers dropped to −27 °F—the lowest temperature ever recorded in Chicago. By contrast, a *local minimum* is a value that's lower than the ones around it, but not necessarily lower than any other value. If the temperature drops a bit one one day, and then raises again on the following day, that's a local minimum.

Glorot initialization

Another name for *Xavier initialization*.

Gradient

Intuitively, the gradient is the "steepness" of a curve at a given point. If the gradient is zero, that means that the curve is flat in that point. If the gradient is a small number, either positive or negative, that means that curve is slightly sloped. If the gradient is a large number, either positive or negative, that means that the curve is very steep. The gradient of a function of one variable, such as `x * 2`, is also called the *derivative*.

Gradient descent

One of the fundamental algorithms of machine learning. Gradient descent iteratively calculates the *gradient* of a system's *loss*, and changes the weights to step in the direction where the loss diminishes. GD is the

subject of Chapter 3, Walking the Gradient, on page 33, and some of its variations are described in Chapter 13, Batchin' Up, on page 157.

Hidden layer

Any layer in a *neural network* that is "inside" the network—that is, neither the *input layer* nor the *output layer*. See Chaining Perceptrons, on page 109.

Hyperparameters

In machine learning, there are two important types of parameters. On one side, there are parameters that the system learns during the *training phase*. On the other, there are parameters that you configure yourself before training—such as the *learning rate*, the number of training iterations, and the number of nodes in the *hidden layers* of a *neural network*. To avoid confusion between the two kinds of parameters, you can use different names: the learnable parameters are often called *weights*, and the parameters that you configure yourself are almost always called "hyperparameters."

Identity function

The function that outputs the same value as its input. It can be used as an *activation function* in some corner cases, as we mention in Picking the Right Function, on page 236.

Input layer

The first *layer* in a *neural network*. It takes the values of the *input variables*. See Chaining Perceptrons, on page 109.

Input variable

See *supervised learning*.

L1/L2

Two common *regularization* techniques, explained in L1 and L2 Regularization, on page 218.

Layers

The main units of organization in a *neural network*. See the diagram in Here's the Plan, on page 113 for an example of a three-layered network. Other entities such as *dropouts* are sometimes called layers—but not all the time.

Leaky ReLU

See *ReLU*.

Learning rate

One of the most important *hyperparameters* of the *gradient descent* algorithm. At each step of GD, the gradient of the weights is multiplied by the learning rate—hence, the bigger the learning rate, the bigger the step.

Learning rate decay

A variant of plain vanilla *gradient descent* where the *learning rate* progressively decreases during training. See Gradient Descent on Steroids, on page 238.

Linear

A function is linear when it can be plotted as a straight shape. Also check out *nonlinear*.

Linear regression

A tecnique that predicts an input from an output by approximating their relation with a line—or with a higher-dimensional *linear* shape. In this book, I use linear regression as the first concrete example of supervised learning, in Chapter 2, Your First Learning Program, on page 15. Later on, in Chapter 4, Hyperspace!, on page 47, I introduce higher-dimensional linear regression.

Linearly separable

A dataset is linearly separable if it can be partitioned with a "straight" shape. For example, imagine a plane populated with two different kinds of data: circles and triangles. If the data is linearly separable, then you can trace a straight line that has all the circles on one side, and all the triangles on the other. Check out Tracing a Boundary, on page 147 for visual examples.

Local gradient

In the context of the *chain rule*, the local gradient is the gradient of an operation's output with respect to its input. I use this concept to explain *backpropagation* in The Chain Rule, on page 129.

Local minimum

See *global minimum*. I talk about both kinds of minima there.

Log loss

One of the formulae that we use to calculate the *loss*. It works well for *binary classification* problems. See Smoothing It Out, on page 69.

Logistic function

A more technically accurate name for the *sigmoid*.

Logistic regression

"Logistic" means "binary" in statistics, and "logistic regression" is a technique to forecast a binary variable. Logistic regression is the theoretical foundation of the *binary classifier* that we build in Chapter 5, A Discerning Machine, on page 65.

Logits

The inputs of a *softmax*.

Loss

A function that measures the error in a machine learning system—in other words, a number that measures how bad the system's forecast is. The *training phase* of a machine learning system is all about minimizing the loss, by tweaking the *weights* with techniques such as *gradient descent*. In this book, we use different formulae to calculate the loss, including the *mean squared error* and the *log loss*.

Matrix

In programming, a matrix is a multidimensional array. In machine learning (and math), it's more specifically a two-dimensional array—a grid.

Matrix multiplication

A frequent operation in machine learning, matrix multiplication operates on two *matrices* of shapes `(a, b)` and `(b, c)`, and results in a third matrix of shape `(a, c)`. Check out the details in Multiplying Matrices, on page 51.

Matrix transpose

An operation that flips a *matrix* over its diagonal. For an example, see Transposing Matrices, on page 53.

Mean squared error

One of the common formulae used to calculate the *loss*. I introduce it in How Wrong Are We?, on page 22.

Mini-batch gradient descent

A variation of *gradient descent* where each step calculates the gradient on a subset of the training examples, rather than all of them. For example, you might have 10000 training examples, but only feed them to GD in batches of 50. The exact number of examples for each mini-batch is a *hyperparameter* of the system. I introduce batch GD in Batch by Batch, on page 160.

Momentum

A variant of *gradient descent*. See Gradient Descent on Steroids, on page 238.

Multilayer perceptron

An alternate (and somewhat old-fashioned) name for a *neural network*.

Multinomial classifier

A *classifier* that assigns data to many classes, as opposed to a *binary classifier*. For example, if you're deciding which brand a car belongs to, then you're doing multinomial classifications—unless you only have two brands, that is. Chapter 7, The Final Challenge, on page 85 is all about multinomial classification.

Multiple linear regression

Linear regression on a function that takes more than one input variable. Chapter 4, Hyperspace!, on page 47 is dedicated to multiple linear regression.

Neuron

Another name for a *node* in a neural network.

Nonlinear

A function is nonlinear when it cannot be plotted as a straight shape. When we talk about nonlinearity in *neural networks*, that's usually in the context of *activation functions*.

Numerical stability

A computation is "numerically unstable" when it tends to generate values that are so large, or so small, that they cause an overflow or an underflow. For example, the *softmax* that we code in this book is numerically unstable because it can easily generate huge values that are too large for Python to handle. For more, see Numerical Stability, on page 120.

One hot encoding

An encoding technique that converts a set of discrete values to a set of arrays, each containg a single 1 and a bunch of 0s. For example, the array `[1, 2, 3]` could be one hot encoded as `[[1, 0, 0], [0, 1, 0], [0, 0, 1]]`. One hot encoding is useful to encode the labels in certain machine learning problems, for reasons I discuss in One Hot Encoding, on page 87.

Output layer

The final *layer* in a *neural network*. See Chaining Perceptrons, on page 109.

Overfitting

A machine learning system is "overfitting" when it's more accurate on the training data than it is on new, unknown data. For example, a face-recognition system might become good at recognizing faces in the pictures that it's been trained on, and then fail when confronted with new pictures of the same people. Overfitting happens because the system "knows" the training data, so it ends up memorizing the accidental details of that data instead of generalizing out of it. To use a metaphor, the system is acting like a student who memorizes a concept by rote learning, without really understanding it.

Partial derivative

Take a function with more than one variable, for example, the function `a * b`, that has two. The partial derivative is the derivative of that function with respect to only one variable—either `a` or `b`. For more details, see Partial Derivatives, on page 40.

Perceptron

Historically, the perceptron was one of the first working machine learning system. It's also the base for *multilayer perceptrons*—that is, *neural networks*. Chapter 8, The Perceptron, on page 95 is dedicated to the perceptron.

Prediction phase

The prediction phase (as opposed to the *training phase*) is the phase of *supervised learning* where you get your money back—the time when you ask your system to predict a *label* from *unlabeled data*. For example, during a speech recognizer's prediction phase, the system could take a sound and "predict" the vocal command in it. I talk about the two phases of supervised learning in Supervised Learning, on page 6.

Reinforcement learning

A style of machine learning where the algorithm learns by trial and error. I talk briefly about reinforcement learning in Programming vs. Machine Learning, on page 4.

Regularization

A family of approaches to reduce *overfitting*. More precisely, "regularization" is any change that you make to a system like a *neural network* to reduce the distance between its performance on the *training set* and its performance on the *validation set*. For the details, read Regularizing the Model, on page 215 and A Regularization Toolbox, on page 222.

ReLU

One of the most popular *activation functions*, described in Enter the ReLU, on page 233. Also comes in a few variants, like the *Leaky ReLU* and the *softplus*.

RMSprop

A variant of *gradient descent*. See Gradient Descent on Steroids, on page 238.

Saturation

An *activation function* saturates when it receives an input that's outside its "ideal" range, which pushes its *gradient* close to zero. The best way to understand this concept is to see a concrete example of it. Read Dead Neurons, on page 139 for an example of a saturating *sigmoid*.

Scalar

A single number you might represent in your code with a single integer or floating point value. By contrast, a value that's composed of multiple scalars is called a "vector"—but in programming, it's more likely to be called an "array."

Sigmoid

The sigmoid is a function that takes any number and converts it to the range from 0 to 1, excluded. It's often used as an *activation function*. I introduced it in Invasion of the Sigmoids, on page 67.

Softmax

The softmax is a function that takes an array of numerical inputs (called *logits*) and returns an array of the same size, where the inputs have been rescaled to the range from 0 to 1, excluded. It's often used as the last *activation function* in a *neural network*. I introduced it in Enter the Softmax, on page 112.

Softplus

See *ReLU*.

Standard deviation

A measure of how "spread out" a variable is. A variable with a low standard deviation rarely strays too far from its average value. One example from Chapter 15, Let's Do Development, on page 175 is the height of humans compared to the height of plants. The second has higher standard deviation than the first, because plants have wildly different heights, while humans don't: nobody is hundreds of times taller than anyone else.

Standardization

A technique to prepare data before feeding it to a machine learning system. There are different ways to "standardize" data, but the most common one is reshaping the input *features* so that they all have the same range and *standard deviation*. For the details, read Preparing Data, on page 176.

Statistical noise

Here is an example to explain what statistical noise is: every time you weight yourself, you're likely to get a different weight. Some of these differences might be meaningful, and they depend on the fact that you're gaining or losing weight. Other differences might be due to irrelevant factors: you weighted yourself at different times during the day, your scale isn't perfectly precise, and so on. These irrelevant fluctuations in real-world data are called statistical noise. We talk about this concept in our discussion of *overfitting*, in The Causes of Overfitting, on page 210.

Stochastic gradient descent

For most people, "stochastic gradient descent" means "*mini-batch gradient descent* with a batch size of 1." In chapters such as Chapter 13, Batchin' Up, on page 157, I refer to that meaning. However, some machine learning practitioners (and some libraries, including Keras) use "stochastic gradient descent" as a synonym to "mini-batch gradient descent," regardless of the batch size. We'll have to live with the confusion, and hope that one of those meanings fades with time.

Strictly increasing/decreasing

Intuitively, a function is strictly increasing if it always points "upward," and strictly decreasing if it always points "downward." More formally, as the input grows, the value of a strictly increasing function always grows, and the value of a strictly decreasing function always drops. For example, if I say "the amount of food we consume increases with the number of dogs we own," that means that once we get a new dog, we'll need the same or more food—never less food. If we say that "the amount of food *strictly* increases with the number of dogs," that means we'll need more food—never the same or less food.

Supervised learning

A style of machine learning where a system takes a set of *examples* in the form of *input variables* and *labels* and learns the relation between the input variables and the label. For example, the input variables could be the number of reservations at a restaurant, and the label could be the number of pizzas sold; or, the input variables could be X-ray scans, and the label could be a boolean value that tells whether the scan shows

pneumonia. This style of machine learning is called "supervised" because somebody, supposedly a human, took care of preparing and labeling the examples.

Swish

See *ReLU*.

Test set

The subset of *examples* you use to test the system. In general, you shouldn't use the same examples to test and train the system, for reasons I explain in Training vs. Testing, on page 77.

Training set

The subset of examples you use to train the system. In general, you shouldn't reuse them to test the system as well, for reasons I explain in Training vs. Testing, on page 77.

Training phase

The training phase (as opposed to the *prediction phase*) is the phase of *supervised learning* where the system goes through the *examples* and learns the relation between the *input variables* and the *label*. For example, during a speech recognizer's training phase, the system could sort through a large number of sound clips, each labeled with the vocal command in the clip. I talk about the two phases of machine learning in Supervised Learning, on page 6.

Tuple

A Python type that is similar to a list, but immutable. See Collections, on page 256.

Underfitting

Simply put, a machine learning system is "underfitting" when it's not powerful enough to make accurate predictions. For example, imagine trying to predict fluctuations in ice cream sales with linear regression. Ice cream sales are mostly periodical—higher in summer and lower in winter—so it's hard to approximate them with a straight line. If you try, then your predictions aren't likely to be very good in most cases. That's a case of underfitting, where you need a more powerful system to predict that phenomenon. Contrast with *overfitting*.

Unsupervised learning

A style of machine learning where the machine finds patterns in data, even without knowing what that data means. I talk about it in What About Unsupervised Learning?, on page 11.

Validation set

The set of examples that is used to test a system's performance during the development process. This should be different from the *test set*, that is only used at the very end of the process. The reason for that difference is that you don't want your system to *overfit* the test data. For a more detailed explanation, see A Testing Conundrum, on page 171.

Vanishing gradient

A problem that plagued deep neural networks for years. The vanishing gradient causes slower and slower training as you add layers to a network. It was partially solved by switching from *sigmoid*-like *activation functions* to other functions such as _ReLU_s. For the nitty gritties, see The Vanishing Gradient, on page 232.

Variance

In machine learning, you can say that a system has "high variance" when it *overfits* the training data, as I explain in Bias and Variance, on page 215.

Weight

One of the learnable *parameters* in a *perceptron*, or a *neural network*.

Weight Decay

The concept behind some *regularization* technique, such as *L1/L2*. It can be wrapped up like this: keep the *weights* in a *neural network* small, so that the network's generates a smoother function. You'll find the gory mathematical details in How L1 and L2 Work, on page 218.

Weighted sum

You do a weighted sum when you add together a number of elements, each multiplied by a *weight*: `element1 * weight1 + element2 * weight2 + ...`

Xavier initialization

Xavier (or "Glorot") initialization is a method to initialize the *weights* in a *neural network*. It helps reduce problems such as *dead neurons* and *vanishing* or *exploding gradients*. For the details, see Better Weight Initialization, on page 238.

Thank you!

How did you enjoy this book? Please let us know. Take a moment and email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line "Book Feedback."

Ready for your next great Pragmatic Bookshelf book? Come on over to https://pragprog.com and use the coupon code BUYANOTHER2019 to save 30% on your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn't apply to *The Pragmatic Programmer* ebook because it's older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

And thank you for your continued support,

Andy Hunt, Publisher

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

https://pragprog.com/book/pplearn

Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up to Date

https://pragprog.com

Join our announcement mailing list (low volume) or follow us on twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy

https://pragprog.com/news

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/pplearn*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764