Neural Networks and Deep Learning

Michael Nielsen

August 7, 2018

Contents

What this book is about

Neural networks are one of the most beautiful programming paradigms ever invented. In the conventional approach to programming, we tell the computer what to do, breaking big problems up into many small, precisely defined tasks that the computer can easily perform. By contrast, in a neural network we don't tell the computer how to solve our problem. Instead, it learns from observational data, figuring out its own solution to the problem at hand.

Automatically learning from data sounds promising. However, until 2006 we didn't know how to train neural networks to surpass more traditional approaches, except for a few specialized problems. What changed in 2006 was the discovery of techniques for learning in so-called deep neural networks. These techniques are now known as deep learning. They've been developed further, and today deep neural networks and deep learning achieve outstanding performance on many important problems in computer vision, speech recognition, and natural language processing. They're being deployed on a large scale by companies such as Google, Microsoft, and Facebook.

The purpose of this book is to help you master the core concepts of neural networks, including modern techniques for deep learning. After working through the book you will have written code that uses neural networks and deep learning to solve complex pattern recognition problems. And you will have a foundation to use neural networks and deep learning to attack problems of your own devising.

A principle-oriented approach

One conviction underlying the book is that it's better to obtain a solid understanding of the core principles of neural networks and deep learning, rather than a hazy understanding of a long laundry list of ideas. If you've understood the core ideas well, you can rapidly understand other new material. In programming language terms, think of it as mastering the core syntax, libraries and data structures of a new language. You may still only "know" a tiny fraction of the total language - many languages have enormous standard libraries - but new libraries and data structures can be understood quickly and easily.

This means the book is emphatically not a tutorial in how to use some particular neural network library. If you mostly want to learn your way around a library, don't read this book! Find the library you wish to learn, and work through the tutorials and documentation. But be warned. While this has an immediate problem-solving payoff, if you want to understand what's really going on in neural networks, if you want insights that will still be relevant years from now, then it's not enough just to learn some hot library. You need to understand the durable, lasting insights underlying how neural networks work. Technologies come and technologies go, but insight is forever.

A hands-on approach

We'll learn the core principles behind neural networks and deep learning by attacking a concrete problem: the problem of teaching a computer to recognize handwritten digits. This problem is extremely difficult to solve using the conventional approach to programming. And yet, as we'll see, it can be solved pretty well using a simple neural network, with just a few tens of lines of code, and no special libraries. What's more, we'll improve the program through many iterations, gradually incorporating more and more of the core ideas about neural networks and deep learning.

This hands-on approach means that you'll need some programming experience to read the book. But you don't need to be a professional programmer. I've written the code in Python (version 2.7), which, even if you don't program in Python, should be easy to understand with just a little effort. Through the course of the book we will develop a little neural network library, which you can use to experiment and to build understanding. All the code is available for download here. Once you've finished the book, or as you read it, you can easily pick up one of the more feature-complete neural network libraries intended for use in production.

On a related note, the mathematical requirements to read the book are modest. There is some mathematics in most chapters, but it's usually just elementary algebra and plots of functions, which I expect most readers will be okay with. I occasionally use more advanced mathematics, but have structured the material so you can follow even if some mathematical details elude you. The one chapter which uses heavier mathematics extensively is Chapter 2, which requires a little multivariable calculus and linear algebra. If those aren't familiar, I begin Chapter 2 with a discussion of how to navigate the mathematics. If you're finding it really heavy going, you can simply skip to the summary of the chapter's main results. In any case, there's no need to worry about this at the outset.

It's rare for a book to aim to be both principle-oriented and hands-on. But I believe you'll learn best if we build out the fundamental ideas of neural networks. We'll develop living code, not just abstract theory, code which you can explore and extend. This way you'll understand the fundamentals, both in theory and practice, and be well set to add further to your knowledge.

1

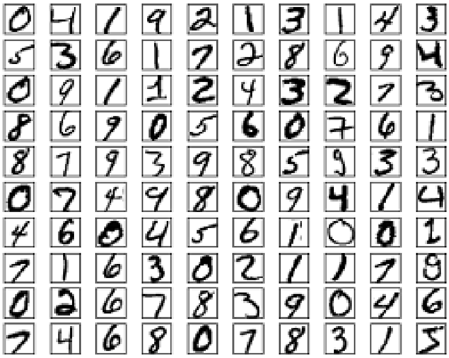
Using neural nets to recognize handwritten digits

The human visual system is one of the wonders of the world. Consider the following sequence of handwritten digits:

Most people effortlessly recognize those digits as 504192. That ease is deceptive. In each hemisphere of our brain, humans have a primary visual cortex, also known as V_1, containing 140 million neurons, with tens of billions of connections between them. And yet human vision involves not just V_1, but an entire series of visual cortices – V_2, V_3, V_4, and V_5 - doing progressively more complex image processing. We carry in our heads a supercomputer, tuned by evolution over hundreds of millions of years, and superbly adapted to understand the visual world. Recognizing handwritten digits isn't easy. Rather, we humans are stupendously, astoundingly good at making sense of what our eyes show us. But nearly all that work is done unconsciously. And so we don't usually appreciate how tough a problem our visual systems solve.

The difficulty of visual pattern recognition becomes apparent if you attempt to write a computer program to recognize digits like those above. What seems easy when we do it ourselves suddenly becomes extremely difficult. Simple intuitions about how we recognize shapes – "a 9 has a loop at the top, and a vertical stroke in the bottom right" – turn out to be not so simple to express algorithmically. When you try to make such rules precise, you quickly get lost in a morass of exceptions and caveats and special cases. It seems hopeless.

Neural networks approach the problem in a different way. The idea is to take a large number of handwritten digits, known as training examples,



and then develop a system which can learn from those training examples. In other words, the neural network uses the examples to automatically infer rules for recognizing handwritten digits. Furthermore, by increasing the number of training examples, the network can learn more about handwriting, and so improve its accuracy. So while I've shown just 100 training digits above, perhaps we could build a better handwriting recognizer by using thousands or even millions or billions of training examples.

In this chapter we'll write a computer program implementing a neural network that learns to recognize handwritten digits. The program is just 74 lines long, and uses no special neural network libraries. But this short program can recognize digits with an accuracy over 96 percent, without human intervention. Furthermore, in later chapters we'll develop ideas which can improve accuracy to over 99 percent. In fact, the best commercial neural networks are now so good that they are used by banks to process cheques, and by post offices to recognize addresses.

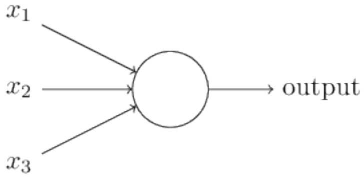
We're focusing on handwriting recognition because it's an excellent prototype problem for learning about neural networks in general. As a prototype it hits a sweet spot: it's challenging – it's no small feat to recognize handwritten digits – but it's not so difficult as to require an extremely complicated solution, or tremendous computational power. Furthermore, it's a great way to develop more advanced techniques, such as deep learning. And so throughout the book we'll return repeatedly to the problem of handwriting recognition. Later in the book, we'll discuss how these ideas may be applied to other problems in computer vision, and also in speech, natural language processing, and other domains.

Of course, if the point of the chapter was only to write a computer program to recognize handwritten digits, then the chapter would be much shorter! But along the way we'll develop many key ideas about neural networks, including two important types of artificial neuron (the perceptron and the sigmoid neuron), and the standard learning algorithm for neural networks, known as stochastic gradient descent. Throughout, I focus on explaining *why* things are done the way they are, and on building your neural networks intuition. That requires a lengthier discussion than if I just presented the basic mechanics of what's going on, but it's worth it for the deeper understanding you'll attain. Amongst the payoffs, by the end of the chapter we'll be in position to understand what deep learning is, and why it matters.

1.1 Perceptrons

What is a neural network? To get started, I'll explain a type of artificial neuron called a *perceptron*. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Today, it's more common to use other models of artificial neurons - in this book, and in much modern work on neural networks, the main neuron model used is one called the *sigmoid neuron*. We'll get to sigmoid neurons shortly. But to understand why sigmoid neurons are defined the way they are, it's worth taking the time to first understand perceptrons.

So how do perceptrons work? A perceptron takes several binary inputs, $x_1, x_2, \ldots$, and produces a single binary output:



In the example shown the perceptron has three inputs, x_1, x_2, x_3. In general it could have more or fewer inputs. Rosenblatt proposed a simple rule to compute the output. He introduced *weights*, w_1,w_2, …, real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some *threshold value*. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms:

$$\text{output} = \begin{cases} 0 & \text{if} \quad \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if} \quad \sum_j w_j x_j > \text{threshold} \end{cases} \tag{1.1}$$

That's all there is to how a perceptron works!

That's the basic mathematical model. A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence. Let me give an example. It's not a very realistic example, but it's easy to understand, and we'll soon get to more realistic examples. Suppose the weekend is coming up, and you've heard that there's going to be a cheese festival in your city. You like cheese, and are trying to decide whether or not to go to the festival. You might make your decision by weighing up three factors:

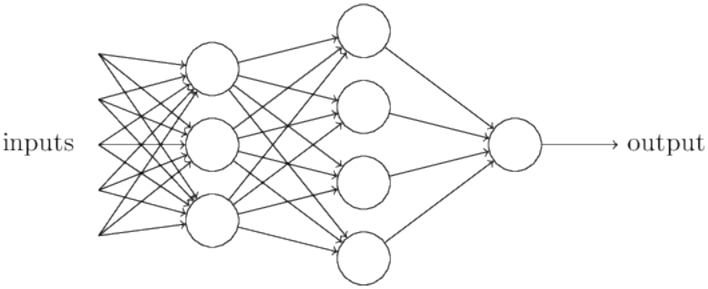
1. Is the weather good?
2. Does your boyfriend or girlfriend want to accompany you?
3. Is the festival near public transit? (You don't own a car).

We can represent these three factors by corresponding binary variables x_1, x_2 and x_3. For instance, we'd have $x_1 = 1$ if the weather is good, and $x_1 = 0$ if the weather is bad. Similarly, $x_2 = 1$ if your boyfriend or girlfriend wants to go, and $x_2 = 0$ if not. And similarly again for x_3 and public transit.

Now, suppose you absolutely adore cheese, so much so that you're happy to go to the festival even if your boyfriend or girlfriend is uninterested and the festival is hard to get to. But perhaps you really loathe bad weather, and there's no way you'd go to the festival if the weather is bad. You can use perceptrons to model this kind of decision-making. One way to do this is to choose a weight $w_1 = 6$ for the weather, and $w_2 = 2$ and $w_3 = 2$ for the other conditions. The larger value of w_1 indicates that the weather matters a lot to you, much more than whether your boyfriend or girlfriend joins you, or the nearness of public transit. Finally, suppose you choose a threshold of 5 for the perceptron. With these choices, the perceptron implements the desired decision-making model, outputting 1 whenever the weather is good, and 0 whenever the weather is bad. It makes no difference to the output whether your boyfriend or girlfriend wants to go, or whether public transit is nearby.

By varying the weights and the threshold, we can get different models of decision-making. For example, suppose we instead chose a threshold of 3. Then the perceptron would decide that you should go to the festival whenever the weather was good or when both the festival was near public transit and your boyfriend or girlfriend was willing to join you. In other words, it'd be a different model of decision-making. Dropping the threshold means you're more willing to go to the festival.

Obviously, the perceptron isn't a complete model of human decision-making! But what the example illustrates is how a perceptron can weigh up different kinds of evidence in order to make decisions. And it should seem plausible that a complex network of perceptrons could make quite subtle decisions:



In this network, the first column of perceptrons–what we'll call the first *layer* of perceptrons–is making three very simple decisions, by weighing the input evidence. What about the perceptrons in the second layer? Each of those perceptrons is making a decision by weighing up the results from the first layer of decision-making. In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer. And even more complex decisions can be made by the perceptron in the third layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

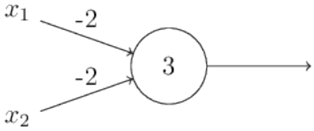
Incidentally, when I defined perceptrons I said that a perceptron has just a single output. In the network above the perceptrons look like they have multiple outputs. In fact, they're still single output. The multiple output arrows are merely a useful way of indicating that the output from a perceptron is being used as the input to several other perceptrons. It's less unwieldy than drawing a single output line which then splits.

Let's simplify the way we describe perceptrons. The condition $\sum_j w_j x_j >$threshold is cumbersome, and we can make two notational changes to simplify it. The first change is to write $\sum_j w_j x_j$ as a dot product, $w \cdot x = \sum_j w_j x_j$, where w and x are vectors whose components are the weights and inputs, respectively. The second change is to move the threshold to the other side of the inequality, and to replace it by what's known as the perceptron's *bias*, b$\equiv -$threshold. Using the bias instead of the threshold, the perceptron rule can be rewritten:

$$\text{output} = \begin{cases} 0 & \text{if} \quad w \cdot x + b \leq 0 \\ 1 & \text{if} \quad w \cdot x + b > 0 \end{cases} \tag{1.2}$$

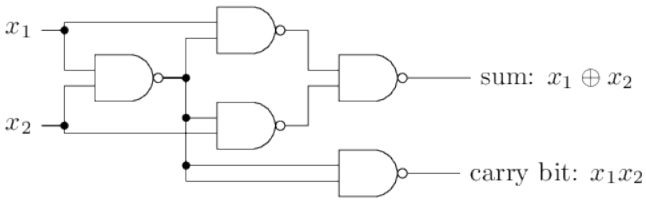
You can think of the bias as a measure of how easy it is to get the perceptron to output a 1. Or to put it in more biological terms, the bias is a measure of how easy it is to get the perceptron to *fire*. For a perceptron with a really big bias, it's extremely easy for the perceptron to output a 1. But if the bias is very negative, then it's difficult for the perceptron to output a 1. Obviously, introducing the bias is only a small change in how we describe perceptrons, but we'll see later that it leads to further notational simplifications. Because of this, in the remainder of the book we won't use the threshold, we'll always use the bias.

I've described perceptrons as a method for weighing evidence to make decisions. Another way perceptrons can be used is to compute the elementary logical functions we usually think of as underlying computation, functions such as AND, OR, and NAND. For example, suppose we have a perceptron with two inputs, each with weight –2, and an overall bias of 3. Here's our perceptron:

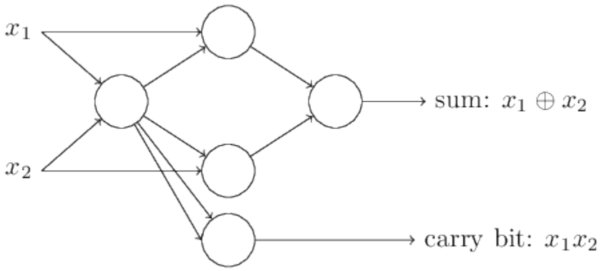


Then we see that input 00 produces output 1, since $(-2)*0+(-2)*0+3=3$ is positive. Here, I've introduced the $*$ symbol to make the multiplications explicit. Similar calculations show that the inputs 01 and 10 produce output 1. But the input 11 produces output 0, since $(-2)*1+(-2)*1+3=-1$ is negative. And so our perceptron implements a NAND gate!

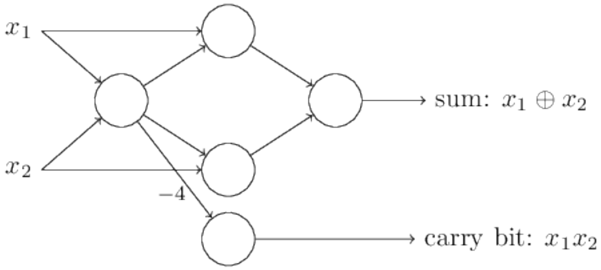
The NAND example shows that we can use perceptrons to compute simple logical functions. In fact, we can use networks of perceptrons to compute *any* logical function at all. The reason is that the NAND gate is universal for computation, that is, we can build any computation up out of NAND gates. For example, we can use NAND gates to build a circuit which adds two bits, x_1 and x_2. This requires computing the bitwise sum, $x_1 \bigoplus x_2$, as well as a carry bit which is set to 1 when both x_1 and x_2 are 1, i.e., the carry bit is just the bitwise product $x_1 x_2$:



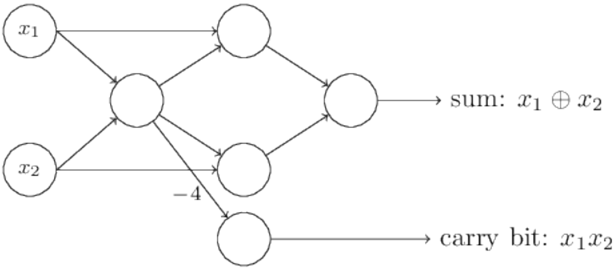
To get an equivalent network of perceptrons we replace all the NAND gates by perceptrons with two inputs, each with weight –2, and an overall bias of 3. Here's the resulting network. Note that I've moved the perceptron corresponding to the bottom right NAND gate a little, just to make it easier to draw the arrows on the diagram:



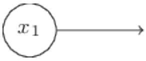
One notable aspect of this network of perceptrons is that the output from the leftmost perceptron is used twice as input to the bottommost perceptron. When I defined the perceptron model I didn't say whether this kind of double-output-to-the-same-place was allowed. Actually, it doesn't much matter. If we don't want to allow this kind of thing, then it's possible to simply merge the two lines, into a single connection with a weight of –4 instead of two connections with –2 weights. (If you don't find this obvious, you should stop and prove to yourself that this is equivalent.) With that change, the network looks as follows, with all unmarked weights equal to –2, all biases equal to 3, and a single weight of –4, as marked:



Up to now I've been drawing inputs like x_1 and x_2 as variables floating to the left of the network of perceptrons. In fact, it's conventional to draw an extra layer of perceptrons – the input layer – to encode the inputs:



This notation for input perceptrons, in which we have an output, but no inputs,



is a shorthand. It doesn't actually mean a perceptron with no inputs. To see this, suppose we did have a perceptron with no inputs. Then the weighted sum $\sum_j w_j x_j$ would always be zero, and so the perceptron would output 1 if $b > 0$, and 0 if $b \leq 0$. That is, the perceptron would simply output a fixed value, not the desired value (x_1, in the example above). It's better to think of the input perceptrons as not really being perceptrons at all, but rather special units which are simply defined to output the desired values, $x_1, x_2, \ldots$.

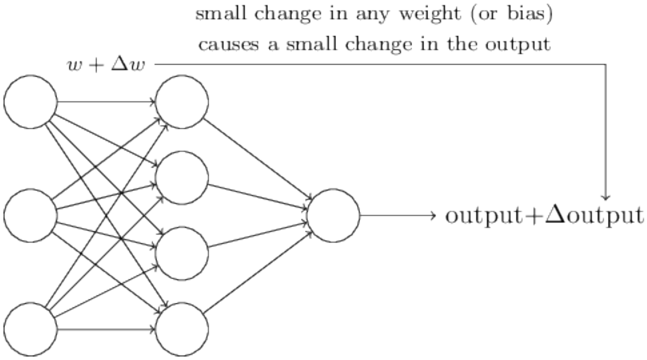
The adder example demonstrates how a network of perceptrons can be used to simulate a circuit containing many NAND gates. And because NAND gates are universal for computation, it follows that perceptrons are also universal for computation.

The computational universality of perceptrons is simultaneously reassuring and disappointing. It's reassuring because it tells us that networks of perceptrons can be as powerful as any other computing device. But it's also disappointing, because it makes it seem as though perceptrons are merely a new type of NAND gate. That's hardly big news!

However, the situation is better than this view suggests. It turns out that we can devise learning algorithms which can automatically tune the weights and biases of a network of artificial neurons. This tuning happens in response to external stimuli, without direct intervention by a programmer. These learning algorithms enable us to use artificial neurons in a way which is radically different to conventional logic gates. Instead of explicitly laying out a circuit of NAND and other gates, our neural networks can simply learn to solve problems, sometimes problems where it would be extremely difficult to directly design a conventional circuit.

1.2 Sigmoid neurons

Learning algorithms sound terrific. But how can we devise such algorithms for a neural network? Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem. For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit. To see how learning might work, suppose we make a small change in some weight (or bias) in the network. What we'd like is for this small change in weight to cause only a small corresponding change in the output from the network. As we'll see in a moment, this property will make learning possible. Schematically, here's what we want (obviously this network is too simple to do handwriting recognition!):

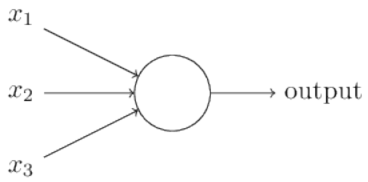

small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want. For example, suppose the network was mistakenly classifying an image as an "8" when it should be a "9". We could figure out how to make a small change in the weights and biases so the network gets a little closer to classifying the image as a "9". And then we'd repeat this, changing the weights and biases over and over to produce better and better output. The network would be learning.

The problem is that this isn't what happens when our network contains perceptrons. In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way. So while your "9" might now be classified correctly, the behaviour of the network on all the other images is likely to have completely changed in some hard-to-control way. That makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to the desired behaviour. Perhaps there's some clever way of getting around this problem. But it's not immediately obvious how we can get a network of perceptrons to learn.

We can overcome this problem by introducing a new type of artificial neuron called a sigmoid neuron. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.

Okay, let me describe the sigmoid neuron. We'll depict sigmoid neurons in the same way we depicted perceptrons:



Just like a perceptron, the sigmoid neuron has inputs, $x_1, x_2, \ldots$. But instead of being just 0 or 1, these inputs can also take on any values between 0 and 1. So, for instance, $0.638 lcdots$ is a valid input for a sigmoid neuron. Also just like a perceptron, the sigmoid neuron has weights for each input, $w_1, w_2, \ldots$, and an overall bias, b. But the output is not 0 or 1. Instead, it's $\sigma(wx + b)$, where σ is called the sigmoid function[1], and is defined by:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}. \tag{1.3}$$

To put it all a little more explicitly, the output of a sigmoid neuron with inputs x_1,x_2,$\ldots$, weights $w_1, w_2, \ldots$, and bias b is

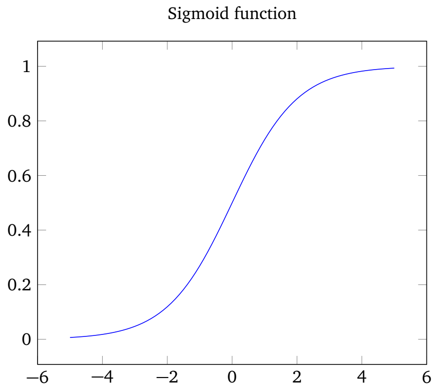
$$\frac{1}{1 + \exp\left(-\sum_j w_j x_j - b\right)}. \tag{1.4}$$

At first sight, sigmoid neurons appear very different to perceptrons. The algebraic form of the sigmoid function may seem opaque and forbidding if you're not already familiar with it. In fact, there are many similarities between perceptrons and sigmoid neurons, and the algebraic form of the sigmoid function turns out to be more of a technical detail than a true barrier to understanding.

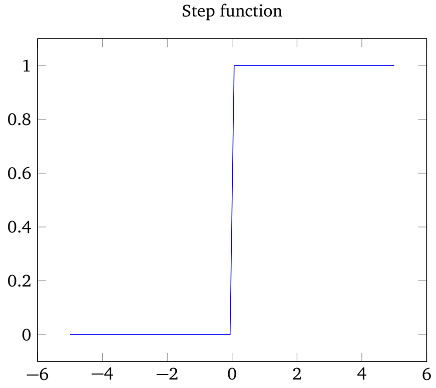
To understand the similarity to the perceptron model, suppose $z \equiv w \cdot x + b$ is a large positive number. Then $e^{-z} \approx 0$ and so $\sigma(z) \approx 1$. In other words, when $z = w \cdot x + b$ is large and positive, the output from the sigmoid neuron is approximately 1, just as it would have been for a perceptron. Suppose on the other hand that$z = w \cdot x + b$ is very negative. Then $e^{-z} \to \infty$, and $\sigma(z) \approx 0$. So when $z = w \cdot x + b$ is very negative, the behaviour of a sigmoid neuron also closely approximates a perceptron. It's only when $w \cdot x + b$ is of modest size that there's much deviation from the perceptron model.

What about the algebraic form of σ? How can we understand that? In fact, the exact form of σ isn't so important – what really matters is the shape of the function when plotted. Here's the shape:

[1] Incidentally, σ is sometimes called the logistic function, and this new class of neurons called logistic neurons. It's useful to remember this terminology, since these terms are used by many people working with neural nets. However, we'll stick with the sigmoid terminology.



This shape is a smoothed out version of a step function:



If σ had in fact been a step function, then the sigmoid neuron would be a perceptron, since the output would be 1 or 0 depending on whether $w \cdot x + b$ was positive or negative[2]. By using the actual σ function we get, as already implied above, a smoothed out perceptron. Indeed, it's the smoothness of the σ function that is the crucial fact, not its detailed form. The smoothness of σ means that small changes Δw_j in the weights and Δb in the bias will produce a small change Δoutput in the output from the neuron. In fact, calculus tells us that Δoutput is well approximated by

$$\Delta\text{output} \approx \sum_j \frac{\partial\,\text{output}}{\partial w_j}\Delta w_j + \frac{\partial\,\text{output}}{\partial b}\Delta b \tag{1.5}$$

where the sum is over all the weights, w_j, and $\partial\,\text{output}/\partial w_j$ and $\partial\,\text{output}/\partial b$ denote partial derivatives of the output with respect to w_j and b, respectively. Don't panic if you're not comfortable with partial derivatives! While the expression above looks complicated, with all the partial derivatives, it's actually saying something very simple (and which is very good news): Δoutput is a linear function of the changes Δw_j and Δb in the weights and bias. This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output. So while sigmoid neurons have much of the same qualitative behavior as perceptrons, they make it much easier to figure out how changing

[2] Actually, when $w \cdot x + b = 0$ the perceptron outputs 0, while the step function outputs 1. So, strictly speaking, we'd need to modify the step function at that one point. But you get the idea.

the weights and biases will change the output.

If it's the shape of σ which really matters, and not its exact form, then why use the particular form used for σ in Equation 1.3? In fact, later in the book we will occasionally consider neurons where the output is $f(w \cdot x + b)$ for some other *activation function* $f(\cdot)$. The main thing that changes when we use a different activation function is that the particular values for the partial derivatives in Equation 1.5 change. It turns out that when we compute those partial derivatives later, using σ will simplify the algebra, simply because exponentials have lovely properties when differentiated. In any case, σ is commonly-used in work on neural nets, and is the activation function we'll use most often in this book.

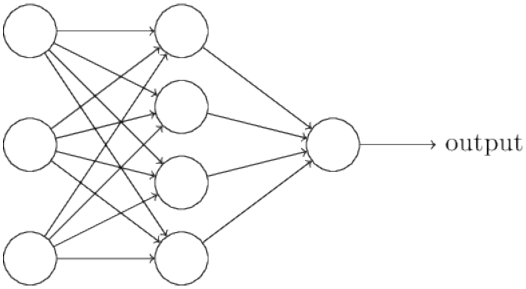
How should we interpret the output from a sigmoid neuron? Obviously, one big difference between perceptrons and sigmoid neurons is that sigmoid neurons don't just output 0 or 1. They can have as output any real number between 0 and 1, so values such as $0.173\ldots$ and $0.689\ldots$ are legitimate outputs. This can be useful, for example, if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network. But sometimes it can be a nuisance. Suppose we want the output from the network to indicate either "the input image is a 9" or "the input image is not a 9". Obviously, it'd be easiest to do this if the output was a 0 or a 1, as in a perceptron. But in practice we can set up a convention to deal with this, for example, by deciding to interpret any output of at least 0.5 as indicating a "9", and any output less than 0.5 as indicating "not a 9". I'll always explicitly state when we're using such a convention, so it shouldn't cause any confusion.

Exercises

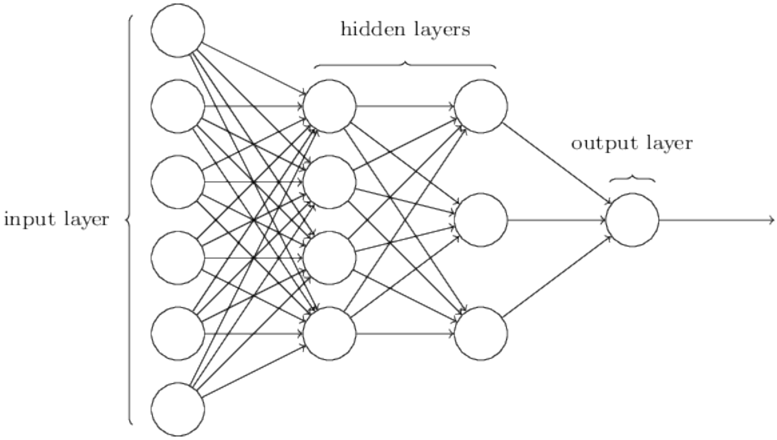
- **Sigmoid neurons simulating perceptrons, part I** Suppose we take all the weights and biases in a network of perceptrons, and multiply them by a positive constant, c>0. Show that the behavior of the network doesn't change.
- **Sigmoid neurons simulating perceptrons, part II** Suppose we have the same setup as the last problem - a network of perceptrons. Suppose also that the overall input to the network of perceptrons has been chosen. We won't need the actual input value, we just need the input to have been fixed. Suppose the weights and biases are such that $w \cdot x + b \neq 0$ for the input x to any particular perceptron in the network. Now replace all the perceptrons in the network by sigmoid neurons, and multiply the weights and biases by a positive constant $c > 0$. Show that in the limit as $c \to \infty$ the behaviour of this network of sigmoid neurons is exactly the same as the network of perceptrons. How can this fail when $w \cdot x + b = 0$ for one of the perceptrons?

1.3 The architecture of neural networks

In the next section I'll introduce a neural network that can do a pretty good job classifying handwritten digits. In preparation for that, it helps to explain some terminology that lets us name different parts of a network. Suppose we have the network:



As mentioned earlier, the leftmost layer in this network is called the input layer, and the neurons within the layer are called *input neurons*. The rightmost or *output* layer contains the *output neurons*, or, as in this case, a single output neuron. The middle layer is called a *hidden layer*, since the neurons in this layer are neither inputs nor outputs. The term “hidden” perhaps sounds a little mysterious – the first time I heard the term I thought it must have some deep philosophical or mathematical significance - but it really means nothing more than “not an input or an output”. The network above has just a single hidden layer, but some networks have multiple hidden layers. For example, the following four-layer network has two hidden layers:



Somewhat confusingly, and for historical reasons, such multiple layer networks are sometimes called *multilayer perceptrons* or *MLPs*, despite being made up of sigmoid neurons, not perceptrons. I’m not going to use the MLP terminology in this book, since I think it’s confusing, but wanted to warn you of its existence.

The design of the input and output layers in a network is often straightforward. For example, suppose we’re trying to determine whether a handwritten image depicts a “9” or not. A natural way to design the network is to encode the intensities of the image pixels into the input neurons. If the image is a 64 by 64 greyscale image, then we’d have $4,096 = 64 \times 64$ input neurons, with the intensities scaled appropriately between 0 and 1. The output layer will contain just a single neuron, with output values of less than 0.5 indicating “input image is not a 9”, and values greater than 0.5 indicating “input image is a 9”.

While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers. In particular, it’s not possible to sum up the design process for the hidden layers with a few simple rules of thumb.

Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behaviour they want out of their nets. For example, such heuristics can be used to help determine how to trade off the number of hidden layers against the time required to train the network. We'll meet several such design heuristics later in this book.

Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer. Such networks are called *feedforward* neural networks. This means there are no loops in the network - information is always fed forward, never fed back. If we did have loops, we'd end up with situations where the input to the σ function depended on the output. That'd be hard to make sense of, and so we don't allow such loops.

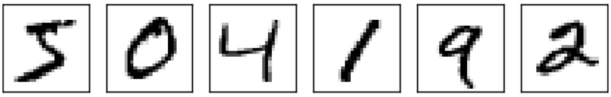
However, there are other models of artificial neural networks in which feedback loops are possible. These models are called recurrent neural networks. The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent. That firing can stimulate other neurons, which may fire a little while later, also for a limited duration. That causes still more neurons to fire, and so over time we get a cascade of neurons firing. Loops don't cause problems in such a model, since a neuron's output only affects its input at some later time, not instantaneously.

Recurrent neural nets have been less influential than feedforward networks, in part because the learning algorithms for recurrent nets are (at least to date) less powerful. But recurrent networks are still extremely interesting. They're much closer in spirit to how our brains work than feedforward networks. And it's possible that recurrent networks can solve important problems which can only be solved with great difficulty by feedforward networks. However, to limit our scope, in this book we're going to concentrate on the more widely-used feedforward networks.

1.4 A simple network to classify handwritten digits

Having defined neural networks, let's return to handwriting recognition. We can split the problem of recognizing handwritten digits into two sub-problems. First, we'd like a way of breaking an image containing many digits into a sequence of separate images, each containing a single digit. For example, we'd like to break the image

into six separate images,

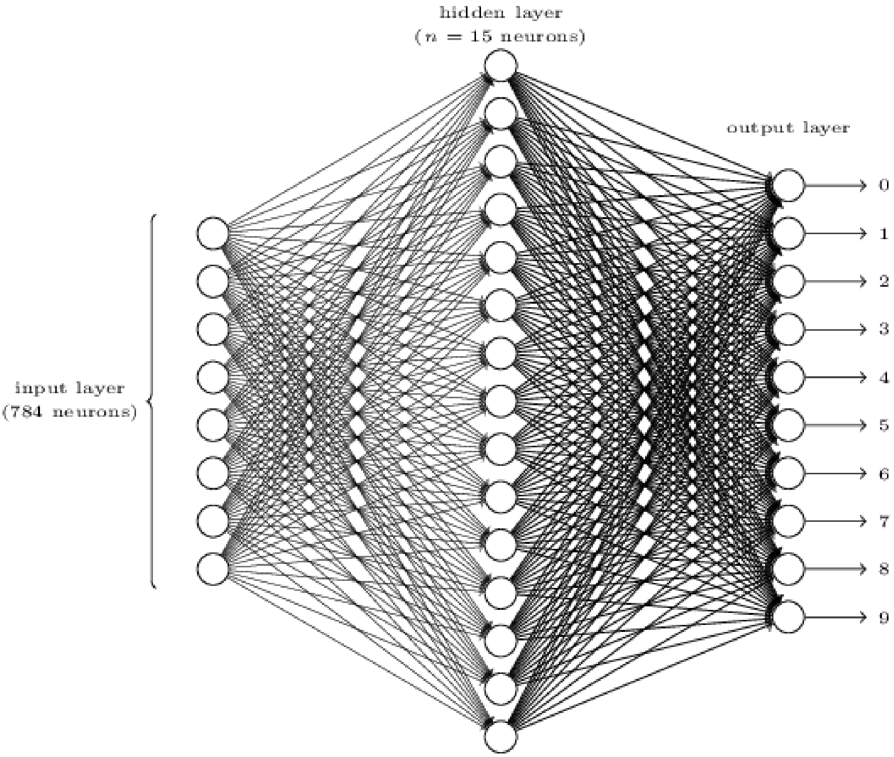


We humans solve this *segmentation problem* with ease, but it's challenging for a computer program to correctly break up the image. Once the image has been segmented, the program then needs to classify each individual digit. So, for instance, we'd like our program to recognize that the first digit above,

is a 5.

We'll focus on writing a program to solve the second problem, that is, classifying individual digits. We do this because it turns out that the segmentation problem is not so difficult to solve, once you have a good way of classifying individual digits. There are many approaches to solving the segmentation problem. One approach is to trial many different ways of segmenting the image, using the individual digit classifier to score each trial segmentation. A trial segmentation gets a high score if the individual digit classifier is confident of its classification in all segments, and a low score if the classifier is having a lot of trouble in one or more segments. The idea is that if the classifier is having trouble somewhere, then it's probably having trouble because the segmentation has been chosen incorrectly. This idea and other variations can be used to solve the segmentation problem quite well. So instead of worrying about segmentation we'll concentrate on developing a neural network which can solve the more interesting and difficult problem, namely, recognizing individual handwritten digits.

To recognize individual digits we will use a three-layer neural network:



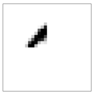
The input layer of the network contains neurons encoding the values of the input pixels. As discussed in the next section, our training data for the network will consist of many 28 by 28 pixel images of scanned handwritten digits, and so the input layer contains $784 = 28 \times 28$ neurons. For simplicity I've omitted most of the 784 input neurons in the diagram above. The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0 representing black, and in between values representing gradually darkening shades of grey.

The second layer of the network is a hidden layer. We denote the number of neurons in this hidden layer by n, and we'll experiment with different values for n. The example shown illustrates a small hidden layer, containing just $n = 15$ neurons.

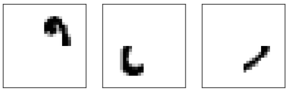
The output layer of the network contains 10 neurons. If the first neuron fires, i.e., has an output ≈ 1, then that will indicate that the network thinks the digit is a 0. If the second neuron fires then that will indicate that the network thinks the digit is a 1. And so on. A little more precisely, we number the output neurons from 0 through 9, and figure out which neuron has the highest activation value. If that neuron is, say, neuron number 6, then our network will guess that the input digit was a 6. And so on for the other output neurons.

You might wonder why we use 10 output neurons. After all, the goal of the network is to tell us which digit (0,1,2,…,9) corresponds to the input image. A seemingly natural way of doing that is to use just 4 output neurons, treating each neuron as taking on a binary value, depending on whether the neuron's output is closer to 0 or to 1. Four neurons are enough to encode the answer, since $2^4 = 16$ is more than the 10 possible values for the input digit. Why should our network use 10 neurons instead? Isn't that inefficient? The ultimate justification is empirical: we can try out both network designs, and it turns out that, for this particular problem, the network with 10 output neurons learns to recognize digits better than the network with 4 output neurons. But that leaves us wondering why using 10 output neurons works better. Is there some heuristic that would tell us in advance that we should use the 10-output encoding instead of the 4-output encoding?

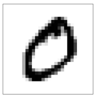
To understand why we do this, it helps to think about what the neural network is doing from first principles. Consider first the case where we use 10 output neurons. Let's concentrate on the first output neuron, the one that's trying to decide whether or not the digit is a 0. It does this by weighing up evidence from the hidden layer of neurons. What are those hidden neurons doing? Well, just suppose for the sake of argument that the first neuron in the hidden layer detects whether or not an image like the following is present:



It can do this by heavily weighting input pixels which overlap with the image, and only lightly weighting the other inputs. In a similar way, let's suppose for the sake of argument that the second, third, and fourth neurons in the hidden layer detect whether or not the following images are present:



As you may have guessed, these four images together make up the 0 image that we saw in the line of digits shown earlier:



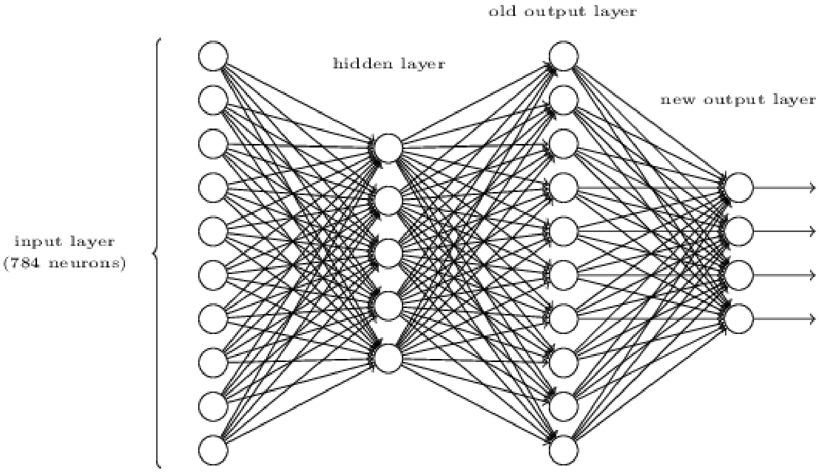
So if all four of these hidden neurons are firing then we can conclude that the digit is a 0. Of course, that's not the only sort of evidence we can use to conclude that the image was a 0 - we could legitimately get a 0 in many other ways (say, through translations of the above images, or slight distortions). But it seems safe to say that at least in this case we'd conclude that the input was a 0.

Supposing the neural network functions in this way, we can give a plausible explanation for why it's better to have 10 outputs from the network, rather than 4. If we had 4 outputs, then the first output neuron would be trying to decide what the most significant bit of the digit was. And there's no easy way to relate that most significant bit to simple shapes like those shown above. It's hard to imagine that there's any good historical reason the component shapes of the digit will be closely related to (say) the most significant bit in the output.

Now, with all that said, this is all just a heuristic. Nothing says that the three-layer neural network has to operate in the way I described, with the hidden neurons detecting simple component shapes. Maybe a clever learning algorithm will find some assignment of weights that lets us use only 4 output neurons. But as a heuristic the way of thinking I've described works pretty well, and can save you a lot of time in designing good neural network architectures.

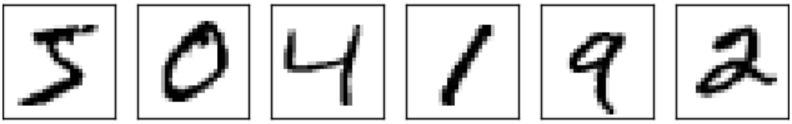
Exercise

- There is a way of determining the bitwise representation of a digit by adding an extra layer to the three-layer network above. The extra layer converts the output from the previous layer into a binary representation, as illustrated in the figure below. Find a set of weights and biases for the new output layer. Assume that the first 3 layers of neurons are such that the correct output in the third layer (i.e., the old output layer) has activation at least 0.99, and incorrect outputs have activation less than 0.01.



1.5 Learning with gradient descent

Now that we have a design for our neural network, how can it learn to recognize digits? The first thing we'll need is a data set to learn from - a so-called training data set. We'll use the MNIST data set, which contains tens of thousands of scanned images of handwritten digits, together with their correct classifications. MNIST's name comes from the fact that it is a modified subset of two data sets collected by NIST, the United States' National Institute of Standards and Technology. Here's a few images from MNIST:



As you can see, these digits are, in fact, the same as those shown at the beginning of this chapter as a challenge to recognize. Of course, when testing our network we'll ask it to recognize images which aren't in the training set!

The MNIST data comes in two parts. The first part contains 60,000 images to be used as training data. These images are scanned handwriting samples from 250 people, half of whom were US Census Bureau employees, and half of whom were high school students. The images are greyscale and 28 by 28 pixels in size. The second part of the MNIST data set is 10,000 images to be used as test data. Again, these are 28 by 28 greyscale images. We'll use the test data to evaluate how well our neural network has learned to recognize digits. To make this a good test of performance, the test data was taken from a different set of 250 people than the original training data (albeit still a group split between Census Bureau employees and high school students). This helps give us confidence that our system can recognize digits from people whose writing it didn't see during training.

We'll use the notation x to denote a training input. It'll be convenient to regard each training input x as a $28 \times 28 = 784$-dimensional vector. Each entry in the vector represents the grey value for a single pixel in the image. We'll denote the corresponding desired output by $y = y(x)$, where y is a 10-dimensional vector. For example, if a particular training image, x, depicts a 6, then $y(x) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$ is the desired output from the network. Note that T here is the transpose operation, turning a row vector into an ordinary (column) vector.

What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x. To quantify how well we're achieving this goal we define a cost function[3]:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2 \tag{1.6}$$

Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x. Of course, the output a depends on x, w and b, but to keep the notation simple I haven't explicitly indicated this dependence. The notation $\|v\|$ just denotes the usual length function for a vector v. We'll call C the *quadratic* cost function; it's also sometimes known as the *mean squared error* or just MSE. Inspecting the form of the quadratic cost function, we see that $C(w, b)$ is non-negative, since every term in the sum is non-negative. Furthermore, the cost $C(w, b)$ becomes small, i.e., $C(w, b) \approx 0$, precisely when $y(x)$ is approximately equal to the output, a, for all training inputs, x. So our training algorithm has done a good job if it can find weights and biases so that $C(w, b) \approx 0$. By contrast, it's not doing so well when $C(w, b)$ is large – that would mean that $y(x)$ is not close to the output a for a large number of inputs. So the aim of our training algorithm will be to minimize the cost $C(w, b)$ as a function of the weights and biases. In other words, we want to find a set of weights and biases which make the cost as small as possible. We'll do that using an algorithm known as *gradient descent*.

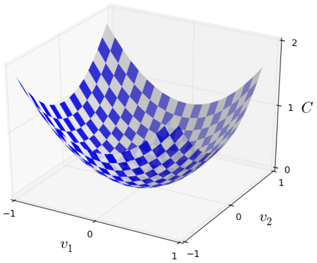
[3] Sometimes referred to as a loss or objective function. We use the term cost function throughout this book, but you should note the other terminology, since it's often used in research papers and other discussions of neural networks.

Why introduce the quadratic cost? After all, aren't we primarily interested in the number of images correctly classified by the network? Why not try to maximize that number directly, rather than minimizing a proxy measure like the quadratic cost? The problem with that is that the number of images correctly classified is not a smooth function of the weights and biases in the network. For the most part, making small changes to the weights and biases won't cause any change at all in the number of training images classified correctly. That makes it difficult to figure out how to change the weights and biases to get improved performance. If we instead use a smooth cost function like the quadratic cost it turns out to be easy to figure out how to make small changes in the weights and biases so as to get an improvement in the cost. That's why we focus first on minimizing the quadratic cost, and only after that will we examine the classification accuracy.

Even given that we want to use a smooth cost function, you may still wonder why we choose the quadratic function used in Equation 1.6. Isn't this a rather ad hoc choice? Perhaps if we chose a different cost function we'd get a totally different set of minimizing weights and biases? This is a valid concern, and later we'll revisit the cost function, and make some modifications. However, the quadratic cost function of Equation 1.6 works perfectly well for understanding the basics of learning in neural networks, so we'll stick with it for now.

Recapping, our goal in training a neural network is to find weights and biases which minimize the quadratic cost function $C(w, b)$. This is a well-posed problem, but it's got a lot of distracting structure as currently posed - the interpretation of w and b as weights and biases, the ÏČ function lurking in the background, the choice of network architecture, MNIST, and so on. It turns out that we can understand a tremendous amount by ignoring most of that structure, and just concentrating on the minimization aspect. So for now we're going to forget all about the specific form of the cost function, the connection to neural networks, and so on. Instead, we're going to imagine that we've simply been given a function of many variables and we want to minimize that function. We're going to develop a technique called gradient descent which can be used to solve such minimization problems. Then we'll come back to the specific function we want to minimize for neural networks.

Okay, let's suppose we're trying to minimize some function, $C(v)$. This could be any real-valued function of many variables, $v = v1, v2, \ldots$. Note that I've replaced the w and b notation by v to emphasize that this could be any function – we're not specifically thinking in the neural networks context any more. To minimize $C(v)$ it helps to imagine C as a function of just two variables, which we'll call v_1 and v_2:



What we'd like is to find where C achieves its global minimum. Now, of course, for the function plotted above, we can eyeball the graph and find the minimum. In that sense, I've perhaps shown slightly too simple a function! A general function, C, may be a complicated function of many variables, and it won't usually be possible to just eyeball the graph to find the minimum.

One way of attacking the problem is to use calculus to try to find the minimum analytically. We could compute derivatives and then try using them to find places where C is an extremum. With some luck that might work when C is a function of just one or a few variables. But it'll turn into a nightmare when we have many more variables. And for neural networks we'll often want far more variables – the biggest neural networks have cost functions which depend on billions of weights and biases in an extremely complicated way. Using calculus to minimize that just won't work!

(After asserting that we'll gain insight by imagining C as a function of just two variables, I've turned around twice in two paragraphs and said, "hey, but what if it's a function of many more than two variables?" Sorry about that. Please believe me when I say that it really does help to imagine C as a function of two variables. It just happens that sometimes that picture breaks down, and the last two paragraphs were dealing with such breakdowns. Good thinking about mathematics often involves juggling multiple intuitive pictures, learning when it's appropriate to use each picture, and when it's not.)

Okay, so calculus doesn't work. Fortunately, there is a beautiful analogy which suggests an algorithm which works pretty well. We start by thinking of our function as a kind of a valley. If you squint just a little at the plot above, that shouldn't be too hard. And we imagine a ball rolling down the slope of the valley. Our everyday experience tells us that the ball will eventually roll to the bottom of the valley. Perhaps we can use this idea as a way to find a minimum for the function? We'd randomly choose a starting point for an (imaginary) ball, and then simulate the motion of the ball as it rolled down to the bottom of the valley. We could do this simulation simply by computing derivatives (and perhaps some second derivatives) of C – those derivatives would tell us everything we need to know about the local "shape" of the valley, and therefore how our ball should roll.

Based on what I've just written, you might suppose that we'll be trying to write down Newton's equations of motion for the ball, considering the effects of friction and gravity, and so on. Actually, we're not going to take the ball-rolling analogy quite that seriously - we're devising an algorithm to minimize C, not developing an accurate simulation of the laws of physics! The ball's-eye view is meant to stimulate our imagination, not constrain our thinking. So rather than get into all the messy details of physics, let's simply ask ourselves: if we were declared God for a day, and could make up our own laws of physics, dictating to the ball how it should roll, what law or laws of motion could we pick that would make it so the ball always rolled to the bottom of the valley?

To make this question more precise, let's think about what happens when we move the ball a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction. Calculus tells us that C changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2. \tag{1.7}$$

We're going to find a way of choosing Δv_1 and Δv_2 so as to make ΔC negative; i.e., we'll choose them so the ball is rolling down into the valley. To figure out how to make such a choice it helps to define Δv to be the vector of changes in v, $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$, where T is again the transpose operation, turning row vectors into column vectors. We'll also define the *gradient* of C to be the vector of partial derivatives, $\left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$. We denote the gradient vector by ∇C, i.e.:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T. \tag{1.8}$$

In a moment we'll rewrite the change ΔC in terms of Δv and the gradient, ∇C. Before getting to that, though, I want to clarify something that sometimes gets people hung up on the gradient. When meeting the ∇C notation for the first time, people sometimes wonder how they should think about the ∇ symbol. What, exactly, does ∇C mean? In fact, it's perfectly fine to think of ∇C as a single mathematical object - the vector defined above - which happens to be written using two symbols. In this point of view, ∇C is just a piece of notational flag-waving, telling you "hey, ∇C is a gradient vector". There are more advanced points of view where ∇C can be viewed as an independent mathematical entity in its own right (for example, as a differential operator), but we won't need such points of view.

With these definitions, the expression 1.7 for ΔC can be rewritten as

$$\Delta C \approx \nabla C \cdot \Delta v \tag{1.9}$$

This equation helps explain why ∇C is called the gradient vector: ∇C relates changes in v to changes in C, just as we'd expect something called a gradient to do. But what's really exciting about the equation is that it lets us see how to choose Δv so as to make ΔC negative. In particular, suppose we choose

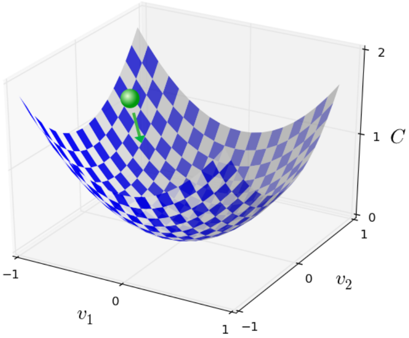
$$\Delta v = -\eta \nabla C, \tag{1.10}$$

where η is a small, positive parameter (known as the *learning rate*). Then Equation 1.9 tells us that $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$. Because $\|\nabla C\|^2 \geq 0$, this guarantees that $\Delta C \leq 0$, i.e., C will always decrease, never increase, if we change v according to the prescription in 1.10. (Within, of course, the limits of the approximation in Equation 1.9). This is exactly the property we wanted! And so we'll take Equation 1.10 to define the "law of motion" for the ball in our gradient descent algorithm. That is, we'll use Equation 1.10 to compute a value for Δv, then move the ball's position v by that amount:

$$v \rightarrow v' = v - \eta \nabla C. \tag{1.11}$$

Then we'll use this update rule again, to make another move. If we keep doing this, over and over, we'll keep decreasing C until - we hope - we reach a global minimum.

Summing up, the way the gradient descent algorithm works is to repeatedly compute the gradient ∇C, and then to move in the opposite direction, "falling down" the slope of the valley. We can visualize it like this:



Notice that with this rule gradient descent doesn't reproduce real physical motion. In real life a ball has momentum, and that momentum may allow it to roll across the slope, or even (momentarily) roll uphill. It's only after the effects of friction set in that the ball is guaranteed to roll down into the valley. By contrast, our rule for choosing Δv just says "go down, right now". That's still a pretty good rule for finding the minimum!

To make gradient descent work correctly, we need to choose the learning rate η to be small enough that Equation 1.9 is a good approximation. If we don't, we might end up with $\Delta C > 0$, which obviously would not be good! At the same time, we don't want η to be too small, since that will make the changes Δv tiny, and thus the gradient descent algorithm will work very slowly. In practical implementations, η is often varied so that Equation 1.9 remains a good approximation, but the algorithm isn't too slow. We'll see later how this works.

I've explained gradient descent when C is a function of just two variables. But, in fact, everything works just as well even when C is a function of many more variables. Suppose in particular that C is a function of m variables, $v_1, \ldots, v_m$. Then the change ΔC in C produced by a small change $\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$ is

$$\Delta C \approx \nabla C \cdot \Delta v, \tag{1.12}$$

where the gradient ∇C is the vector

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m} \right)^T. \tag{1.13}$$

Just as for the two variable case, we can choose

$$\Delta v = -\eta \nabla C, \tag{1.14}$$

and we're guaranteed that our (approximate) expression 1.12 for ΔC will be negative. This gives us a way of following the gradient to a minimum, even when C is a function of many variables, by repeatedly applying the update rule

$$v \rightarrow v' = v - \eta \nabla C. \tag{1.15}$$

You can think of this update rule as *defining* the gradient descent algorithm. It gives us a way of repeatedly changing the position v in order to find a minimum of the function C. The rule doesn't always work - several things can go wrong and prevent gradient descent from finding the global minimum of C, a point we'll return to explore in later chapters. But, in practice gradient descent often works extremely well, and in neural networks we'll find that it's a powerful way of minimizing the cost function, and so helping the net learn.

Indeed, there's even a sense in which gradient descent is the optimal strategy for searching for a minimum. Let's suppose that we're trying to make a move Δv in position so as to decrease C as much as possible. This is equivalent to minimizing $\Delta C \approx \nabla C \cdot \Delta v$. We'll constrain the size of the move so that $\|\Delta v\| = \epsilon$ for some small fixed $\epsilon > 0$. In other words, we want a move that is a small step of a fixed size, and we're trying to find the movement direction which decreases C as much as possible. It can be proved that the choice of Δv which minimizes $\nabla C \cdot \Delta v$ is $\Delta v = -\eta \nabla C$, where $\eta = \epsilon / \|\nabla C\|$ is determined by the size constraint $\|\Delta v\| = \epsilon$. So gradient descent can be viewed as a way of taking small steps in the direction which does the most to immediately decrease C.

Exercises

- Prove the assertion of the last paragraph. Hint: If you're not already familiar with the Cauchy-Schwarz inequality, you may find it helpful to familiarize yourself with it.
- I explained gradient descent when C is a function of two variables, and when it's a function of more than two variables. What happens when C is a function of just one variable? Can you provide a geometric interpretation of what gradient descent is doing in the one-dimensional case?

People have investigated many variations of gradient descent, including variations that more closely mimic a real physical ball. These ball-mimicking variations have some advantages, but also have a major disadvantage: it turns out to be necessary to compute second partial derivatives of C, and this can be quite costly. To see why it's costly, suppose we want to compute all the second partial derivatives $\partial^2 C / \partial v_j \partial v_k$. If there are a million such v_j variables then we'd need to compute something like a trillion (i.e., a million squared) second partial derivatives[4]! That's going to be computationally costly. With that said, there are tricks for avoiding this kind of problem, and finding alternatives to gradient descent is an active area of investigation. But in this book we'll use gradient descent (and variations) as our main approach to learning in neural networks.

How can we apply gradient descent to learn in a neural network? The idea is to use gradient descent to find the weights wk and biases bl which minimize the cost in Equation 1.6. To see how this works, let's restate the gradient descent update rule, with the weights and biases replacing the variables v_j. In other words, our "position" now has components w_k and b_l, and the gradient vector ∇C has corresponding components $\partial C / \partial w_k$ and $\partial C / \partial b_l$. Writing out the gradient descent update rule in terms of components, we have

$$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{1.16}$$

$$b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}. \tag{1.17}$$

By repeatedly applying this update rule we can "roll down the hill", and hopefully find a minimum of the cost function. In other words, this is a rule which can be used to learn in a neural network.

[4] Actually, more like half a trillion, since $\partial^2 C / \partial v_j \partial v_k = \partial^2 C / \partial v_k \partial v_j$. Still, you get the point.

There are a number of challenges in applying the gradient descent rule. We'll look into those in depth in later chapters. But for now I just want to mention one problem. To understand what the problem is, let's look back at the quadratic cost in Equation 1.6. Notice that this cost function has the form $C = \frac{1}{n}\sum_x C_x$, that is, it's an average over costs $C_x \equiv \frac{\|y(x)-a\|^2}{2}$ for individual training examples. In practice, to compute the gradient ∇C we need to compute the gradients ∇C_xseparately for each training input, x, and then average them, $\nabla C = \frac{1}{n}\sum_x \nabla C_x$. Unfortunately, when the number of training inputs is very large this can take a long time, and learning thus occurs slowly.

An idea called *stochastic gradient descent* can be used to speed up learning. The idea is to estimate the gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs. By averaging over this small sample it turns out that we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning.

To make these ideas more precise, stochastic gradient descent works by randomly picking out a small number m of randomly chosen training inputs. We'll label those random training inputs $X_1, X_2, \ldots, X_m$, and refer to them as a mini-batch. Provided the sample size m is large enough we expect that the average value of the ∇C_{X_j}will be roughly equal to the average over all ∇C_x, that is,

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C, \tag{1.18}$$

where the second sum is over the entire set of training data. Swapping sides we get

$$\nabla C \approx \frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j}, \tag{1.19}$$

confirming that we can estimate the overall gradient by computing gradients just for the randomly chosen mini-batch.

To connect this explicitly to learning in neural networks, suppose wk and bl denote the weights and biases in our neural network. Then stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs, and training with those,

$$w_k \quad \rightarrow \quad w_k' = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{1.20}$$

$$b_l \quad \rightarrow \quad b_l' = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l}, \tag{1.21}$$

where the sums are over all the training examples X_j in the current mini-batch. Then we pick out another randomly chosen mini-batch and train with those. And so on, until we've exhausted the training inputs, which is said to complete an *epoch* of training. At that point we start over with a new training epoch.

Incidentally, it's worth noting that conventions vary about scaling of the cost function and of mini-batch updates to the weights and biases. In Equation 1.6 we scaled the overall cost function by a factor $\frac{1}{n}$. People sometimes omit the $\frac{1}{n}$, summing over the costs of individual training examples instead of averaging. This is particularly useful when the total number of training examples isn't known in advance. This can occur if more training data is being generated in real time, for instance. And, in a similar way, the mini-batch update rules 1.20 and 1.21 sometimes omit the $\frac{1}{m}$ term out the front of the sums. Conceptually this makes little difference, since it's equivalent to rescaling the learning rate η. But when doing detailed comparisons of different work it's worth watching out for.

We can think of stochastic gradient descent as being like political polling: it's much easier to sample a small mini-batch than it is to apply gradient descent to the full batch, just as carrying out a poll is easier than running a full election. For example, if we have a training set of size n=60,000, as in MNIST, and choose a mini-batch size of (say) $m = 10$, this means we'll get a factor of 6,000 speedup in estimating the gradient! Of course, the estimate won't be perfect - there will be statistical fluctuations - but it doesn't need to be perfect: all we really care about is moving in a general direction that will help decrease C, and that means we don't need an exact computation of the gradient. In practice, stochastic gradient descent is a commonly used and powerful technique for learning in neural networks, and it's the basis for most of the learning techniques we'll develop in this book.

Exercize

- An extreme version of gradient descent is to use a mini-batch size of just 1. That is, given a training input, x, we update our weights and biases according to the rules $w_k \rightarrow w'_k = w_k - \eta \partial C_x / \partial w_k$ and $b_l \rightarrow b'_l = b_l - \eta \partial C_x / \partial b_l$. Then we choose another training input, and update the weights and biases again. And so on, repeatedly. This procedure is known as online, on-line, or incremental learning. In online learning, a neural network learns from just one training input at a time (just as human beings do). Name one advantage and one disadvantage of online learning, compared to stochastic gradient descent with a mini-batch size of, say, 20.

Let me conclude this section by discussing a point that sometimes bugs people new to gradient descent. In neural networks the cost C is, of course, a function of many variables - all the weights and biases - and so in some sense defines a surface in a very high-dimensional space. Some people get hung up thinking: "Hey, I have to be able to visualize all these extra dimensions". And they may start to worry: "I can't think in four dimensions, let alone five (or five million)". Is there some special ability they're missing, some ability that "real" supermathematicians have? Of course, the answer is no. Even most professional mathematicians can't visualize four dimensions especially well, if at all. The trick they use, instead, is to develop other ways of representing what's going on. That's exactly what we did above: we used an algebraic (rather than visual) representation of ΔC to figure out how to move so as to decrease C. People who are good at thinking in high dimensions have a mental library containing many different techniques along these lines; our algebraic trick is just one example. Those techniques may not have the simplicity we're accustomed to when visualizing three dimensions, but once you build up a library of such techniques, you can get pretty good at thinking in high dimensions. I won't go into more detail here, but if you're interested then you may enjoy reading this discussion of some of the techniques professional mathematicians use to think in high dimensions. While some of the techniques discussed are quite complex, much of the best content is intuitive and accessible, and could be mastered by anyone.

1.6 Implementing our network to classify digits

Alright, let's write a program that learns how to recognize handwritten digits, using stochastic gradient descent and the MNIST training data. We'll do this with a short Python (2.7) program, just 74 lines of code! The first thing we need is to get the MNIST data. If you're a git user then you can obtain the data by cloning the code repository for this book,

```
git clone https://github.com/mnielsen/neural-networks-and-deep-learning.git
```

If you don't use git then you can download the data and code [link]here.

Incidentally, when I described the MNIST data earlier, I said it was split into 60,000 training images, and 10,000 test images. That's the official MNIST description. Actually, we're going to split the data a little differently. We'll leave the test images as is, but split the 60,000-image MNIST training set into two parts: a set of 50,000 images, which we'll use to train our neural network, and a separate 10,000 image validation set. We won't use the validation data in this chapter, but later in the book we'll find it useful in figuring out how to set certain hyper-parameters of the neural network - things like the learning rate, and so on, which aren't directly selected by our learning algorithm. Although the validation data isn't part of the original MNIST specification, many people use MNIST in this fashion, and the use of validation data is common in neural networks. When I refer to the "MNIST training data" from now on, I'll be referring to our 50,000 image data set, not the original 60,000 image data set[5].

Apart from the MNIST data we also need a Python library called Numpy, for doing fast linear algebra. If you don't already have Numpy installed, you can get it [link]here.

Let me explain the core features of the neural networks code, before giving a full listing, below. The centerpiece is a `Network` class, which we use to represent a neural network. Here's the code we use to initialize a Network object:

```
class Network(object):
  def __init__(self, sizes):
    self.num_layers = len(sizes)
    self.sizes = sizes
    self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
    self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]
```

In this code, the list `sizes` contains the number of neurons in the respective layers. So, for example, if we want to create a `Network` object with 2 neurons in the first layer, 3 neurons in the second layer, and 1 neuron in the final layer, we'd do this with the code:

```
net = Network([2, 3, 1])
```

The biases and weights in the `Network` object are all initialized randomly, using the Numpy `np.random.randn` function to generate Gaussian distributions with mean 0 and standard deviation 1. This random initialization gives our stochastic gradient descent algorithm a place to start from. In later chapters we'll find better ways of initializing the weights and biases, but this will do for now. Note that the Network initialization code assumes that the first layer of neurons is an input layer, and omits to set any biases for those neurons, since biases are only ever used in computing the outputs from later layers.

Note also that the biases and weights are stored as lists of Numpy matrices. So, for example `net.weights[1]` is a Numpy matrix storing the weights connecting the second and third layers of neurons. (It's not the first and second layers, since Python's list indexing starts at 0.) Since `net.weights[1]` is rather verbose, let's just denote that matrix w. It's a matrix such that w_{jk} is the weight for the connection between the k^{th} neuron in the second layer, and the j^{th} neuron in the third layer. This ordering of the j and k indices may seem strange – surely it'd make more sense to swap the j and k indices around? The big advantage of using

[5] As noted earlier, the MNIST data set is based on two data sets collected by NIST, the United States' National Institute of Standards and Technology. To construct MNIST the NIST data sets were stripped down and put into a more convenient format by Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. See this [link] for more details. The data set in my repository is in a form that makes it easy to load and manipulate the MNIST data in Python. I obtained this particular form of the data from the LISA machine learning laboratory at the University of Montreal ([link]).

this ordering is that it means that the vector of activations of the third layer of neurons is:

$$a' = \sigma(wa + b). \tag{1.22}$$

There's quite a bit going on in this equation, so let's unpack it piece by piece. a is the vector of activations of the second layer of neurons. To obtain a' we multiply a by the weight matrix w, and add the vector b of biases. We then apply the function σ elementwise to every entry in the vector $wa + b$. (This is called *vectorizing* the function σ.) It's easy to verify that Equation 1.22 gives the same result as our earlier rule, Equation 1.4, for computing the output of a sigmoid neuron.

Exercise

- Write out Equation 1.22 in component form, and verify that it gives the same result as the rule 1.4 for computing the output of a sigmoid neuron.

With all this in mind, it's easy to write code computing the output from a Network instance. We begin by defining the sigmoid function:

```
def sigmoid(z):
  return 1.0/(1.0+np.exp(-z))
```

Note that when the input `z` is a vector or Numpy array, Numpy automatically applies the function `sigmoid` elementwise, that is, in vectorized form.

We then add a `feedforward` method to the `Network` class, which, given an input a for the network, returns the corresponding output[6]. All the method does is applies Equation 1.22 for each layer:

```
def feedforward(self, a):
  """Return the output of the network if "a" is input."""
  for b, w in zip(self.biases, self.weights):
    a = sigmoid(np.dot(w, a)+b)
  return a
```

Of course, the main thing we want our `Network` objects to do is to learn. To that end we'll give them an SGD method which implements stochastic gradient descent. Here's the code. It's a little mysterious in a few places, but I'll break it down below, after the listing.

```
def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
  """Train the neural network using mini-batch stochastic gradient descent.  The "
     training_data" is a list of tuples "(x, y)" representing the training inputs and the
      desired outputs.  The other non-optional parameters are self-explanatory.  If "
     test_data" is provided then the network will be evaluated against the test data
     after each epoch, and partial progress printed out.  This is useful for tracking
     progress, but slows things down substantially."""
  if test_data:
    n_test = len(test_data)
    n = len(training_data)
  for j in xrange(epochs):
    random.shuffle(training_data)
    mini_batches = [training_data[k:k+mini_batch_size] for k in xrange(0, n,
      mini_batch_size)]
    for mini_batch in mini_batches:
      self.update_mini_batch(mini_batch, eta)
  if test_data:
    print "Epoch {0}: {1} / {2}".format(j, self.evaluate(test_data), n_test)
  else:
    print "Epoch {0} complete".format(j)
```

[6] It is assumed that the input a is an (n, 1) Numpy ndarray, not a (n,) vector. Here, n is the number of inputs to the network. If you try to use an (n,) vector as input you'll get strange results. Although using an (n,) vector appears the more natural choice, using an (n, 1) ndarray makes it particularly easy to modify the code to feedforward multiple inputs at once, and that is sometimes convenient.

The `training_data` is a list of tuples `(x, y)` representing the training inputs and corresponding desired outputs. The variables `epochs` and `mini_batch_size` are what you'd expect – the number of epochs to train for, and the size of the mini-batches to use when sampling. `eta` is the learning rate, η. If the optional argument `test_data` is supplied, then the program will evaluate the network after each epoch of training, and print out partial progress. This is useful for tracking progress, but slows things down substantially.

The code works as follows. In each epoch, it starts by randomly shuffling the training data, and then partitions it into mini-batches of the appropriate size. This is an easy way of sampling randomly from the training data. Then for each `mini_batch` we apply a single step of gradient descent. This is done by the code `self.update_mini_batch(mini_batch, eta)`, which updates the network weights and biases according to a single iteration of gradient descent, using just the training data in `mini_batch`. Here's the code for the `update_mini_batch` method:

```
def update_mini_batch(self, mini_batch, eta):
  """Update the network's weights and biases by applying gradient descent using
     backpropagation to a single mini batch. The "mini_batch" is a list of tuples "(x, y)
     ", and "eta" is the learning rate."""
  nabla_b = [np.zeros(b.shape) for b in self.biases]
  nabla_w = [np.zeros(w.shape) for w in self.weights]
  for x, y in mini_batch:
    delta_nabla_b, delta_nabla_w = self.backprop(x, y)
    nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
    nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
  self.weights = [w-(eta/len(mini_batch))*nw for w, nw in zip(self.weights, nabla_w)]
  self.biases = [b-(eta/len(mini_batch))*nb for b, nb in zip(self.biases, nabla_b)]
```

Most of the work is done by the line

```
delta_nabla_b, delta_nabla_w = self.backprop(x, y)
```

This invokes something called the *backpropagation* algorithm, which is a fast way of computing the gradient of the cost function. So `update_mini_batch` works simply by computing these gradients for every training example in the `mini_batch`, and then updating `self.weights` and `self.biases` appropriately.

I'm not going to show the code for `self.backprop` right now. We'll study how backpropagation works in the next chapter, including the code for `self.backprop`. For now, just assume that it behaves as claimed, returning the appropriate gradient for the cost associated to the training example x.

Let's look at the full program, including the documentation strings, which I omitted above. Apart from `self.backprop` the program is self-explanatory – all the heavy lifting is done in `self.SGD` and `self.update_mini_batch`, which we've already discussed. The `self.backprop` method makes use of a few extra functions to help in computing the gradient, namely `sigmoid_prime`, which computes the derivative of the σ function, and `self.cost_derivative`, which I won't describe here. You can get the gist of these (and perhaps the details) just by looking at the code and documentation strings. We'll look at them in detail in the next chapter. Note that while the program appears lengthy, much of the code is documentation strings intended to make the code easy to understand. In fact, the program contains just 74 lines of non-whitespace, non-comment code. All the code may be found on GitHub [link]here.

```
"""
network.py
~~~~~~~~~~
A module to implement the stochastic gradient descent learning
algorithm for a feedforward neural network.  Gradients are calculated
using backpropagation.  Note that I have focused on making the code
simple, easily readable, and easily modifiable.  It is not optimized,
and omits many desirable features.
```

```
"""
#### Libraries
# Standard library
import random
# Third-party libraries
import numpy as np
class Network(object):
  def __init__(self, sizes):
    """The list ``sizes`` contains the number of neurons in the
    respective layers of the network.  For example, if the list
    was [2, 3, 1] then it would be a three-layer network, with the
    first layer containing 2 neurons, the second layer 3 neurons,
    and the third layer 1 neuron.  The biases and weights for the
    network are initialized randomly, using a Gaussian
    distribution with mean 0, and variance 1.  Note that the first
    layer is assumed to be an input layer, and by convention we
    won't set any biases for those neurons, since biases are only
    ever used in computing the outputs from later layers."""
    self.num_layers = len(sizes)
    self.sizes = sizes
    self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
    self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

  def feedforward(self, a):
    """Return the output of the network if ``a`` is input."""
    for b, w in zip(self.biases, self.weights):
    a = sigmoid(np.dot(w, a)+b)
    return a

  def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
    """Train the neural network using mini-batch stochastic
    gradient descent.  The ``training_data`` is a list of tuples
    ``(x, y)`` representing the training inputs and the desired
    outputs.  The other non-optional parameters are
    self-explanatory.  If ``test_data`` is provided then the
    network will be evaluated against the test data after each
    epoch, and partial progress printed out.  This is useful for
    tracking progress, but slows things down substantially."""
    if test_data:
      n_test = len(test_data)
    n = len(training_data)
    for j in xrange(epochs):
      random.shuffle(training_data)
      mini_batches = [
        training_data[k:k+mini_batch_size]
        for k in xrange(0, n, mini_batch_size)]
      for mini_batch in mini_batches:
        self.update_mini_batch(mini_batch, eta)
      if test_data:
        print "Epoch {0}: {1} / {2}".format(
          j, self.evaluate(test_data), n_test)
      else:
        print "Epoch {0} complete".format(j)

  def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch.
    The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
    is the learning rate."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
      delta_nabla_b, delta_nabla_w = self.backprop(x, y)
      nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
      nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
            for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
            for b, nb in zip(self.biases, nabla_b)]

  def backprop(self, x, y):
    """Return a tuple ``(nabla_b, nabla_w)`` representing the
    gradient for the cost function C_x.  ``nabla_b`` and
    ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
    to ``self.biases`` and ``self.weights``."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
      z = np.dot(w, activation)+b
      zs.append(z)
      activation = sigmoid(z)
      activations.append(activation)
    # backward pass
    delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    # Note that the variable l in the loop below is used a little
    # differently to the notation in Chapter 2 of the book. Here,
    # l = 1 means the last layer of neurons, l = 2 is the
    # second-last layer, and so on. It's a renumbering of the
    # scheme in the book, used here to take advantage of the fact
    # that Python can use negative indices in lists.
    for l in xrange(2, self.num_layers):
      z = zs[-l]
      sp = sigmoid_prime(z)
      delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
      nabla_b[-l] = delta
      nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)

  def evaluate(self, test_data):
    """Return the number of test inputs for which the neural
    network outputs the correct result. Note that the neural
    network's output is assumed to be the index of whichever
    neuron in the final layer has the highest activation."""
    test_results = [(np.argmax(self.feedforward(x)), y)
            for (x, y) in test_data]
    return sum(int(x == y) for (x, y) in test_results)

  def cost_derivative(self, output_activations, y):
    """Return the vector of partial derivatives \partial C_x /
    \partial a for the output activations."""
    return (output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
  """The sigmoid function."""
  return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
  """Derivative of the sigmoid function."""
  return sigmoid(z)*(1-sigmoid(z))
```

How well does the program recognize handwritten digits? Well, let's start by loading in the MNIST data. I'll do this using a little helper program, `mnist_loader.py`, to be described below. We execute the following commands in a Python shell,

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
```

Of course, this could also be done in a separate Python program, but if you're following along it's probably easiest to do in a Python shell.

After loading the MNIST data, we'll set up a Network with 30 hidden neurons. We do this after importing the Python program listed above, which is named network,

```
>>> import network
>>> net = network.Network([784, 30, 10])
```

Finally, we'll use stochastic gradient descent to learn from the MNIST `training_data` over 30 epochs, with a mini-batch size of 10, and a learning rate of $\eta = 3.0$,

```
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Note that if you're running the code as you read along, it will take some time to execute – for a typical machine (as of 2015) it will likely take a few minutes to run. I suggest you set things running, continue to read, and periodically check the output from the code. If you're in a rush you can speed things up by decreasing the number of epochs, by decreasing the number of hidden neurons, or by using only part of the training data. Note that production code would be much, much faster: these Python scripts are intended to help you understand how neural nets work, not to be high-performance code! And, of course, once we've trained a network it can be run very quickly indeed, on almost any computing platform. For example, once we've learned a good set of weights and biases for a network, it can easily be ported to run in Javascript in a web browser, or as a native app on a mobile device. In any case, here is a partial transcript of the output of one training run of the neural network. The transcript shows the number of test images correctly recognized by the neural network after each epoch of training. As you can see, after just a single epoch this has reached 9,129 out of 10,000, and the number continues to grow,

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

That is, the trained network gives us a classification rate of about 95 percent - 95.42 percent at its peak ("Epoch 28")! That's quite encouraging as a first attempt. I should warn you, however, that if you run the code then your results are not necessarily going to be quite the same as mine, since we'll be initializing our network using (different) random weights and biases. To generate results in this chapter I've taken best-of-three runs.

Let's rerun the above experiment, changing the number of hidden neurons to 100. As was the case earlier, if you're running the code as you read along, you should be warned that it takes quite a while to execute (on my machine this experiment takes tens of seconds for each training epoch), so it's wise to continue reading in parallel while the code executes.

```
>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Sure enough, this improves the results to 96.59 percent. At least in this case, using more hidden neurons helps us get better results[7]

Of course, to obtain these accuracies I had to make specific choices for the number of epochs of training, the mini-batch size, and the learning rate, η. As I mentioned above, these are known as hyper-parameters for our neural network, in order to distinguish them from the parameters (weights and biases) learnt by our learning algorithm. If we choose our hyper-parameters poorly, we can get bad results. Suppose, for example, that we'd chosen the learning rate to be $\eta = 0.001$,

```
>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 0.001, test_data=test_data)
```

The results are much less encouraging,

```
Epoch 0: 1139 / 10000
Epoch 1: 1136 / 10000
Epoch 2: 1135 / 10000
```

[7] Reader feedback indicates quite some variation in results for this experiment, and some training runs give results quite a bit worse. Using the techniques introduced in chapter 3 will greatly reduce the variation in performance across different training runs for our networks.

```
...
Epoch 27: 2101 / 10000
Epoch 28: 2123 / 10000
Epoch 29: 2142 / 10000
```

However, you can see that the performance of the network is getting slowly better over time. That suggests increasing the learning rate, say to $\eta = 0.01$. If we do that, we get better results, which suggests increasing the learning rate again. (If making a change improves things, try doing more!) If we do that several times over, we'll end up with a learning rate of something like $\eta = 1.0$ (and perhaps fine tune to 3.0), which is close to our earlier experiments. So even though we initially made a poor choice of hyper-parameters, we at least got enough information to help us improve our choice of hyper-parameters. In general, debugging a neural network can be challenging. This is especially true when the initial choice of hyper-parameters produces results no better than random noise. Suppose we try the successful 30 hidden neuron network architecture from earlier, but with the learning rate changed to $\eta = 100.0$:

```
>>> net = network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 100.0, test_data=test_data)
```

At this point we've actually gone too far, and the learning rate is too high:

```
Epoch 0: 1009 / 10000
Epoch 1: 1009 / 10000
Epoch 2: 1009 / 10000
Epoch 3: 1009 / 10000
...
Epoch 27: 982 / 10000
Epoch 28: 982 / 10000
Epoch 29: 982 / 10000
```

Now imagine that we were coming to this problem for the first time. Of course, we know from our earlier experiments that the right thing to do is to decrease the learning rate. But if we were coming to this problem for the first time then there wouldn't be much in the output to guide us on what to do. We might worry not only about the learning rate, but about every other aspect of our neural network. We might wonder if we've initialized the weights and biases in a way that makes it hard for the network to learn? Or maybe we don't have enough training data to get meaningful learning? Perhaps we haven't run for enough epochs? Or maybe it's impossible for a neural network with this architecture to learn to recognize handwritten digits? Maybe the learning rate is too low? Or, maybe, the learning rate is too high? When you're coming to a problem for the first time, you're not always sure.

The lesson to take away from this is that debugging a neural network is not trivial, and, just as for ordinary programming, there is an art to it. You need to learn that art of debugging in order to get good results from neural networks. More generally, we need to develop heuristics for choosing good hyper-parameters and a good architecture. We'll discuss all these at length through the book, including how I chose the hyper-parameters above.

Exercise

- Try creating a network with just two layers – an input and an output layer, no hidden layer – with 784 and 10 neurons, respectively. Train the network using stochastic gradient descent. What classification accuracy can you achieve?

Earlier, I skipped over the details of how the MNIST data is loaded. It's pretty straightforward. For completeness, here's the code. The data structures used to store the MNIST data are described in the documentation strings – it's straightforward stuff, tuples and lists of Numpy ndarray objects (think of them as vectors if you're not familiar with ndarrays):

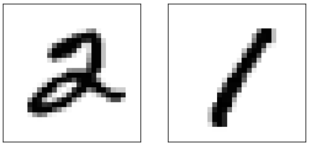
```
"""
mnist_loader
```

```
~~~~~~~~~~~~

A library to load the MNIST image data.  For details of the data
structures that are returned, see the doc strings for ``load_data``
and ``load_data_wrapper``.  In practice, ``load_data_wrapper`` is the
function usually called by our neural network code.
"""

#### Libraries
# Standard library
import cPickle
import gzip

# Third-party libraries
import numpy as np

def load_data():
  """Return the MNIST data as a tuple containing the training data, the validation data,
     and the test data.

  The ``training_data`` is returned as a tuple with two entries. The first entry contains
      the actual training images.  This is a
  numpy ndarray with 50,000 entries.  Each entry is, in turn, a numpy ndarray with 784
     values, representing the 28 * 28 = 784
  pixels in a single MNIST image.

  The second entry in the ``training_data`` tuple is a numpy ndarray  containing 50,000
     entries.  Those entries are just the digit
  values (0...9) for the corresponding images contained in the first  entry of the tuple.

  The ``validation_data`` and ``test_data`` are similar, except   each contains only
     10,000 images.

  This is a nice data format, but for use in neural networks it's helpful to modify the
     format of the ``training_data`` a little.
  That's done in the wrapper function ``load_data_wrapper()``, see below.
  """
  f = gzip.open('../data/mnist.pkl.gz', 'rb')
  training_data, validation_data, test_data = cPickle.load(f)
  f.close()
  return (training_data, validation_data, test_data)

def load_data_wrapper():
  """Return a tuple containing ``(training_data, validation_data,
  test_data)``. Based on ``load_data``, but the format is more
  convenient for use in our implementation of neural networks.

  In particular, ``training_data`` is a list containing 50,000
  2-tuples ``(x, y)``.  ``x`` is a 784-dimensional numpy.ndarray
  containing the input image.  ``y`` is a 10-dimensional
  numpy.ndarray representing the unit vector corresponding to the
  correct digit for ``x``.

  ``validation_data`` and ``test_data`` are lists containing 10,000
  2-tuples ``(x, y)``.  In each case, ``x`` is a 784-dimensional
  numpy.ndarry containing the input image, and ``y`` is the
  corresponding classification, i.e., the digit values (integers)
  corresponding to ``x``.
  Obviously, this means we're using slightly different formats for
  the training data and the validation / test data.  These formats
  turn out to be the most convenient for use in our neural network
  code."""
  tr_d, va_d, te_d = load_data()
  training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
  training_results = [vectorized_result(y) for y in tr_d[1]]
  training_data = zip(training_inputs, training_results)
  validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
  validation_data = zip(validation_inputs, va_d[1])
  test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
  test_data = zip(test_inputs, te_d[1])
  return (training_data, validation_data, test_data)

def vectorized_result(j):
  """Return a 10-dimensional unit vector with a 1.0 in the jth
```

```
position and zeroes elsewhere.  This is used to convert a digit
(0...9) into a corresponding desired output from the neural
network."""
e = np.zeros((10, 1))
e[j] = 1.0
return e
```

I said above that our program gets pretty good results. What does that mean? Good compared to what? It's informative to have some simple (non-neural-network) baseline tests to compare against, to understand what it means to perform well. The simplest baseline of all, of course, is to randomly guess the digit. That'll be right about ten percent of the time. We're doing much better than that!

What about a less trivial baseline? Let's try an extremely simple idea: we'll look at how dark an image is. For instance, an image of a 2 will typically be quite a bit darker than an image of a 1, just because more pixels are blackened out, as the following examples illustrate:



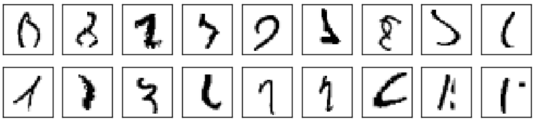
This suggests using the training data to compute average darknesses for each digit, 0,1,2,...,9. When presented with a new image, we compute how dark the image is, and then guess that it's whichever digit has the closest average darkness. This is a simple procedure, and is easy to code up, so I won't explicitly write out the code – if you're interested it's in the [link]GitHub repository. But it's a big improvement over random guessing, getting 2,225 of the 10,000 test images correct, i.e., 22.25 percent accuracy.

It's not difficult to find other ideas which achieve accuracies in the 20 to 50 percent range. If you work a bit harder you can get up over 50 percent. But to get much higher accuracies it helps to use established machine learning algorithms. Let's try using one of the best known algorithms, the *support vector machine* or *SVM*. If you're not familiar with SVMs, not to worry, we're not going to need to understand the details of how SVMs work. Instead, we'll use a Python library called [link]scikit-learn, which provides a simple Python interface to a fast C-based library for SVMs known as [link]LIBSVM.

If we run scikit-learn's SVM classifier using the default settings, then it gets 9,435 of 10,000 test images correct. (The code is available [link]here.) That's a big improvement over our naive approach of classifying an image based on how dark it is. Indeed, it means that the SVM is performing roughly as well as our neural networks, just a little worse. In later chapters we'll introduce new techniques that enable us to improve our neural networks so that they perform much better than the SVM.

That's not the end of the story, however. The 9,435 of 10,000 result is for scikit-learn's default settings for SVMs. SVMs have a number of tunable parameters, and it's possible to search for parameters which improve this out-of-the-box performance. I won't explicitly do this search, but instead refer you to this [link]blog post by [link]Andreas Müller if you'd like to know more. Mueller shows that with some work optimizing the SVM's parameters it's possible to get the performance up above 98.5 percent accuracy. In other words, a well-tuned SVM only makes an error on about one digit in 70. That's pretty good! Can neural networks do better?

In fact, they can. At present, well-designed neural networks outperform every other technique for solving MNIST, including SVMs. The current (2013) record is classifying 9,979 of 10,000 images correctly. This was done by [link]Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus. We'll see most of the techniques they used later in the book. At that level the performance is close to human-equivalent, and is arguably better, since quite a few of the MNIST images are difficult even for humans to recognize with confidence, for example:



I trust you'll agree that those are tough to classify! With images like these in the MNIST data set it's remarkable that neural networks can accurately classify all but 21 of the 10,000 test images. Usually, when programming we believe that solving a complicated problem like recognizing the MNIST digits requires a sophisticated algorithm. But even the neural networks in the Wan et al paper just mentioned involve quite simple algorithms, variations on the algorithm we've seen in this chapter. All the complexity is learned, automatically, from the training data. In some sense, the moral of both our results and those in more sophisticated papers, is that for some problems:

$$\text{sophisticated algorithm} \leq \text{simple learning algorithm} + \text{good training data}.$$

1.7 Toward deep learning

While our neural network gives impressive performance, that performance is somewhat mysterious. The weights and biases in the network were discovered automatically. And that means we don't immediately have an explanation of how the network does what it does. Can we find some way to understand the principles by which our network is classifying handwritten digits? And, given such principles, can we do better?

To put these questions more starkly, suppose that a few decades hence neural networks lead to artificial intelligence (AI). Will we understand how such intelligent networks work? Perhaps the networks will be opaque to us, with weights and biases we don't understand, because they've been learned automatically. In the early days of AI research people hoped that the effort to build an AI would also help us understand the principles behind intelligence and, maybe, the functioning of the human brain. But perhaps the outcome will be that we end up understanding neither the brain nor how artificial intelligence works!

To address these questions, let's think back to the interpretation of artificial neurons that I gave at the start of the chapter, as a means of weighing evidence. Suppose we want to determine whether an image shows a human face or not:

We could attack this problem the same way we attacked handwriting recognition - by using the pixels in the image as input to a neural network, with the output from the network a single neuron indicating either "Yes, it's a face" or "No, it's not a face".

Let's suppose we do this, but that we're not using a learning algorithm. Instead, we're going to try to design a network by hand, choosing appropriate weights and biases. How might we go about it? Forgetting neural networks entirely for the moment, a heuristic we could use is to decompose the problem into sub-problems: does the image have an eye in the top left? Does it have an eye in the top right? Does it have a nose in the middle? Does it have a mouth in the bottom middle? Is there hair on top? And so on.

If the answers to several of these questions are "yes", or even just "probably yes", then we'd conclude that the image is likely to be a face. Conversely, if the answers to most of the questions are "no", then the image probably isn't a face.

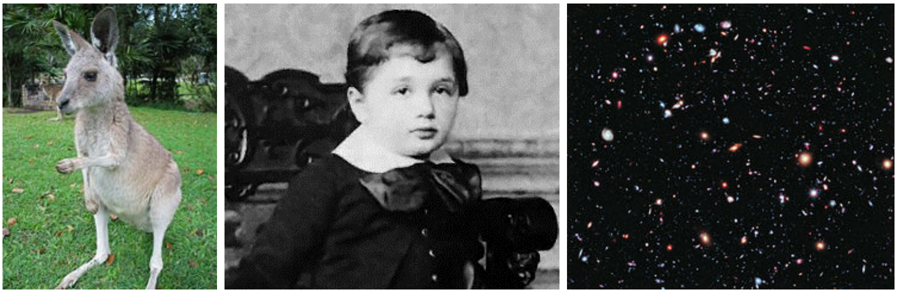
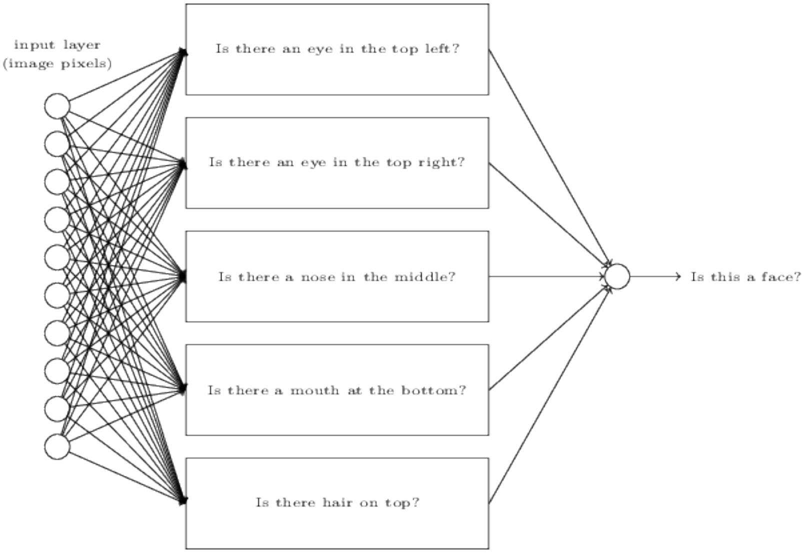
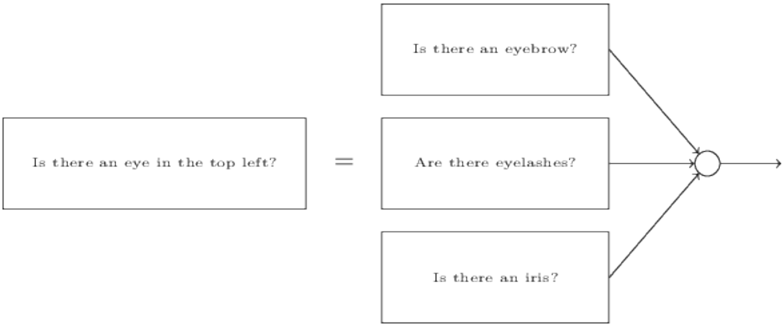


Figure 1.1: Credits: 1. Ester Inbar. 2. Unknown. 3. NASA, ESA, G. Illingworth, D. Magee, and P. Oesch (University of California, Santa Cruz), R. Bouwens (Leiden University), and the HUDF09 Team. [link]Click on the images for more details.es for more details.

Of course, this is just a rough heuristic, and it suffers from many deficiencies. Maybe the person is bald, so they have no hair. Maybe we can only see part of the face, or the face is at an angle, so some of the facial features are obscured. Still, the heuristic suggests that if we can solve the sub-problems using neural networks, then perhaps we can build a neural network for face-detection, by combining the networks for the sub-problems. Here's a possible architecture, with rectangles denoting the sub-networks. Note that this isn't intended as a realistic approach to solving the face-detection problem; rather, it's to help us build intuition about how networks function. Here's the architecture:


input layer
(image pixels)
Is there an eye in the top left?
Is there an eye in the top right?
Is there a nose in the middle?
Is there a mouth at the bottom?
Is there hair on top?
Is this a face?

It's also plausible that the sub-networks can be decomposed. Suppose we're considering the question: "Is there an eye in the top left?" This can be decomposed into questions such as: "Is there an eyebrow?"; "Are there eyelashes?"; "Is there an iris?"; and so on. Of course, these questions should really include positional information, as well - "Is the eyebrow in the top left, and above the iris?", that kind of thing – but let's keep it simple. The network to answer the question "Is there an eye in the top left?" can now be decomposed:



Those questions too can be broken down, further and further through multiple layers. Ultimately, we'll be working with sub-networks that answer questions so simple they can easily be answered at the level of single pixels. Those questions might, for example, be about the presence or absence of very simple shapes at particular points in the image. Such questions can be answered by single neurons connected to the raw pixels in the image.

The end result is a network which breaks down a very complicated question - does this image show a face or not - into very simple questions answerable at the level of single pixels. It does this through a series of many layers, with early layers answering very simple and specific questions about the input image, and later layers building up a hierarchy of ever more complex and abstract concepts. Networks with this kind of many-layer structure – two or more hidden layers – are called *deep neural networks*.

Of course, I haven't said how to do this recursive decomposition into sub-networks. It certainly isn't practical to hand-design the weights and biases in the network. Instead, we'd like to use learning algorithms so that the network can automatically learn the weights and biases - and thus, the hierarchy of concepts - from training data. Researchers in the 1980s and 1990s tried using stochastic gradient descent and backpropagation to train deep networks. Unfortunately, except for a few special architectures, they didn't have much luck. The networks would learn, but very slowly, and in practice often too slowly to be useful.

Since 2006, a set of techniques has been developed that enable learning in deep neural nets. These deep learning techniques are based on stochastic gradient descent and backpropagation, but also introduce new ideas. These techniques have enabled much deeper (and larger) networks to be trained – people now routinely train networks with 5 to 10 hidden layers. And, it turns out that these perform far better on many problems than shallow neural networks, i.e., networks with just a single hidden layer. The reason, of course, is the ability of deep nets to build up a complex hierarchy of concepts. It's a bit like the way conventional programming languages use modular design and ideas about abstraction to enable the creation of complex computer programs. Comparing a deep network to a shallow network is a bit like comparing a programming language with the ability to make function calls to a stripped down language with no ability to make such calls. Abstraction takes a different form in neural networks than it does in conventional programming, but it's just as important.

2 How the backpropagation algorithm works

In the last chapter we saw how neural networks can learn their weights and biases using the gradient descent algorithm. There was, however, a gap in our explanation: we didn't discuss how to compute the gradient of the cost function. That's quite a gap! In this chapter I'll explain a fast algorithm for computing such gradients, an algorithm known as *backpropagation*.

The backpropagation algorithm was originally introduced in the 1970s, but its importance wasn't fully appreciated until a famous 1986 [link]paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams. That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural nets to solve problems which had previously been insoluble. Today, the backpropagation algorithm is the workhorse of learning in neural networks.

This chapter is more mathematically involved than the rest of the book. If you're not crazy about mathematics you may be tempted to skip the chapter, and to treat backpropagation as a black box whose details you're willing to ignore. Why take the time to study those details?

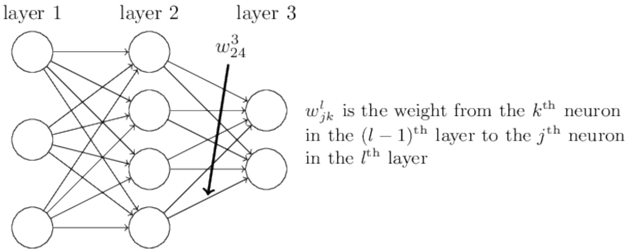
The reason, of course, is understanding. At the heart of backpropagation is an expression for the partial derivative $\partial C/\partial w$ of the cost function C with respect to any weight w (or bias b) in the network. The expression tells us how quickly the cost changes when we change the weights and biases. And while the expression is somewhat complex, it also has a beauty to it, with each element having a natural, intuitive interpretation. And so backpropagation isn't just a fast algorithm for learning. It actually gives us detailed insights into how changing the weights and biases changes the overall behaviour of the network. That's well worth studying in detail.

With that said, if you want to skim the chapter, or jump straight to the next chapter, that's fine. I've written the rest of the book to be accessible even if you treat backpropagation as a black box. There are, of course, points later in the book where I refer back to results from this chapter. But at those points you should still be able to understand the main conclusions, even if you don't follow all the reasoning.

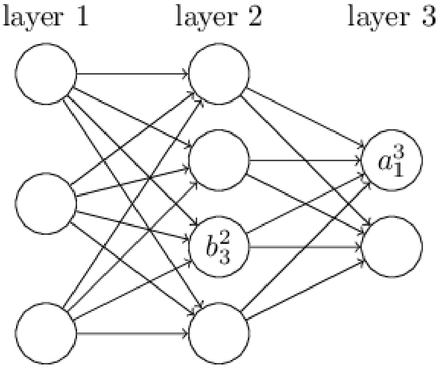
2.1 Warm up: a fast matrix-based approach to computing the output from a neural network

Before discussing backpropagation, let's warm up with a fast matrix-based algorithm to compute the output from a neural network. We actually already briefly saw this algorithm near the end of the last chapter, but I described it quickly, so it's worth revisiting in detail. In particular, this is a good way of getting comfortable with the notation used in backpropagation, in a familiar context.

Let's begin with a notation which lets us refer to weights in the network in an unambiguous way. We'll use w^l_{jk} to denote the weight for the connection from the k-th neuron in the $(l-1)$-th layer to the j-th neuron in the l-th layer. So, for example, the diagram below shows the weight on a connection from the fourth neuron in the second layer to the second neuron in the third layer of a network:


layer 1
layer 2
layer 3
w^3_{24}
w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

This notation is cumbersome at first, and it does take some work to master. But with a little effort you'll find the notation becomes easy and natural. One quirk of the notation is the ordering of the j and k indices. You might think that it makes more sense to use j to refer to the input neuron, and k to the output neuron, not vice versa, as is actually done. I'll explain the reason for this quirk below. We use a similar notation for the network's biases and activations. Explicitly, we use b^l_j for the bias of the j-th neuron in the l-th layer. And we use a^l_j for the activation of the j-th neuron in the l-th layer. The following diagram shows examples of these notations in use:



With these notations, the activation a^l_j of the j-th neuron in the l-th layer is related to the activations in the $(l-1)$-th layer by the equation (compare Equation 1.4 and surrounding discussion in the last chapter)

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right), \tag{2.1}$$

where the sum is over all neurons k in the $(l-1)$-th layer. To rewrite this expression in a matrix form we define a weight matrix w_l for each layer, l. The entries of the weight matrix w_l are just the weights connecting to the l-th layer of neurons, that is, the entry in the j-th row and k-th column is w^l_{jk}. Similarly, for each layer l we define a *bias vector*, b^l. You can probably guess how this works - the components of the bias vector are just the values b^l_j, one component for each neuron in the l-th layer. And finally, we define an activation vector al whose components are the activations a^l_j. The last ingredient we need to rewrite 2.1 in a matrix form is the idea of vectorizing a function such as σ. We met vectorization briefly in the last chapter, but to recap, the idea is that we want to apply a function such as σ to every element in a vector v. We use the obvious notation $\sigma(v)$ to denote this kind of elementwise application of a function. That is, the components of $\sigma(v)$ are just $\sigma(v)_j = \sigma(v_j)$. As an example, if we have the function $f(x) = x^2$ then the vectorized form of f has the effect

$$f\left(\begin{bmatrix} 2 \\ 3 \end{bmatrix}\right) = \begin{bmatrix} f(2) \\ f(3) \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \end{bmatrix}, \tag{2.2}$$

that is, the vectorized f just squares every element of the vector.

With these notations in mind, Equation 2.1 can be rewritten in the beautiful and compact vectorized form

$$a^l = \sigma(w^l a^{l-1} + b^l). \tag{2.3}$$

This expression gives us a much more global way of thinking about how the activations in one layer relate to activations in the previous layer: we just apply the weight matrix to the activations, then add the bias vector, and finally apply the σ function[1]. That global view is often easier and more succinct (and involves fewer indices!) than the neuron-by-neuron view we've taken to now. Think of it as a way of escaping index hell, while remaining precise about what's going on. The expression is also useful in practice, because most matrix libraries provide fast ways of implementing matrix multiplication, vector addition, and vectorization. Indeed, the code in the last chapter made implicit use of this expression to compute the behaviour of the network.

When using Equation 2.3 to compute a^l, we compute the intermediate quantity $z^l \equiv w^l a^{l-1} + b^l$ along the way. This quantity turns out to be useful enough to be worth naming: we call z^l the weighted input to the neurons in layer l. We'll make considerable use of the weighted input z^l later in the chapter. Equation 2.3 is sometimes written in terms of the weighted input, as $a^l = \sigma(z^l)$. It's also worth noting that z^l has components $z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$, that is, z^l_j is just the weighted input to the activation function for neuron j in layer l.

[1] By the way, it's this expression that motivates the quirk in the w^l_{jk} notation mentioned earlier. If we used j to index the input neuron, and k to index the output neuron, then we'd need to replace the weight matrix in Equation 2.3 by the transpose of the weight matrix. That's a small change, but annoying, and we'd lose the easy simplicity of saying (and thinking) "apply the weight matrix to the activations".

2.2 The two assumptions we need about the cost function

The goal of backpropagation is to compute the partial derivatives $\partial C/\partial w$ and $\partial C/\partial b$ of the cost function C with respect to any weight w or bias b in the network. For backpropagation to work we need to make two main assumptions about the form of the cost function. Before stating those assumptions, though, it's useful to have an example cost function in mind. We'll use the quadratic cost function from last chapter (c.f. Equation 1.6). In the notation of the last section, the quadratic cost has the form

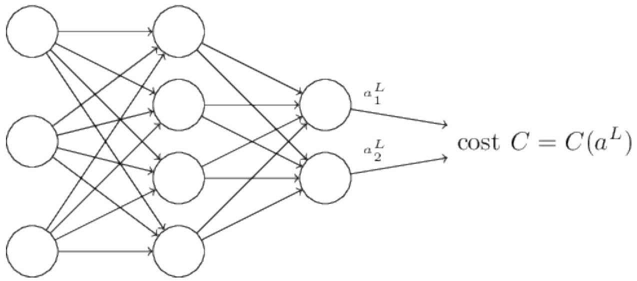
$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2, \tag{2.4}$$

where: n is the total number of training examples; the sum is over individual training examples, x; $y = y(x)$ is the corresponding desired output; L denotes the number of layers in the network; and $a^L = a^L(x)$ is the vector of activations output from the network when x is input.

Okay, so what assumptions do we need to make about our cost function, C, in order that backpropagation can be applied? The first assumption we need is that the cost function can be written as an average $C = \frac{1}{n} \sum_x C_x$ over cost functions C_x for individual training examples, x. This is the case for the quadratic cost function, where the cost for a single training example is $C_x = \frac{1}{2}\|y - a^L\|^2$. This assumption will also hold true for all the other cost functions we'll meet in this book.

The reason we need this assumption is because what backpropagation actually lets us do is compute the partial derivatives $\partial C_x/\partial w$ and $\partial C_x/\partial b$ for a single training example. We then recover $\partial C/\partial w$ and $\partial C/\partial b$ by averaging over training examples. In fact, with this assumption in mind, we'll suppose the training example x has been fixed, and drop the x subscript, writing the cost C_x as C. We'll eventually put the x back in, but for now it's a notational nuisance that is better left implicit.

The second assumption we make about the cost is that it can be written as a function of the outputs from the neural network:



For example, the quadratic cost function satisfies this requirement, since the quadratic cost for a single training example x may be written as

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2} \sum_j (y_j - a^L_j)^2, \tag{2.5}$$

and thus is a function of the output activations. Of course, this cost function also depends on the desired output y, and you may wonder why we're not regarding the cost also as a function of y. Remember, though, that the input training example x is fixed, and so the output y is also a fixed parameter. In particular, it's not something we can modify by changing the weights and biases in any way, i.e., it's not something which the neural network learns. And so it makes sense to regard C as a function of the output activations a^L alone, with y merely a parameter that helps define that function.

2.3 The Hadamard product, $s \odot t$

The backpropagation algorithm is based on common linear algebraic operations - things like vector addition, multiplying a vector by a matrix, and so on. But one of the operations is a little less commonly used. In particular, suppose s and t are two vectors of the same dimension. Then we use $s \odot t$ to denote the elementwise product of the two vectors. Thus the components of $s \odot t$ are just $(s \odot t)_j = s_j t_j$. As an example,

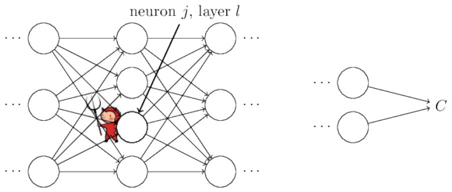
$$\left[\begin{array}{c} 1 \\ 2 \end{array}\right] \odot \left[\begin{array}{c} 3 \\ 4 \end{array}\right] = \left[\begin{array}{c} 1 * 3 \\ 2 * 4 \end{array}\right] = \left[\begin{array}{c} 3 \\ 8 \end{array}\right]. \tag{2.6}$$

This kind of elementwise multiplication is sometimes called the *Hadamard product* or *Schur product*. We'll refer to it as the Hadamard product. Good matrix libraries usually provide fast implementations of the Hadamard product, and that comes in handy when implementing backpropagation.

2.4 The four fundamental equations behind backpropagation

Backpropagation is about understanding how changing the weights and biases in a network changes the cost function. Ultimately, this means computing the partial derivatives $\partial C/\partial w^l_{jk}$ and $\partial C/\partial b^l_j$. But to compute those, we first introduce an intermediate quantity, δ^l_j, which we call the *error* in the j-th neuron in the l-th layer. Backpropagation will give us a procedure to compute the error δ^l_j, and then will relate δ^l_j to $\partial C/\partial w^l_{jk}$ and $\partial C/\partial b^l_j$.

To understand how the error is defined, imagine there is a demon in our neural network:



The demon sits at the j-th neuron in layer l. As the input to the neuron comes in, the demon messes with the neuron's operation. It adds a little change δz^l_j to the neuron's weighted input, so that instead of outputting $\sigma(z^l_j)$, the neuron instead outputs $\sigma(z^l_j + \delta z^l_j)$. This change propagates through later layers in the network, finally causing the overall cost to change by an amount $\partial C \partial z^l_j \delta z^l_j$.

Now, this demon is a good demon, and is trying to help you improve the cost, i.e., they're trying to find a δz^l_j which makes the cost smaller. Suppose $\partial C/\partial z^l_j$ has a large value (either positive or negative). Then the demon can lower the cost quite a bit by choosing δz^l_j to have the opposite sign to $\partial C/\partial z^l_j$. By contrast, if $\partial C/\partial z^l_j$ is close to zero, then the demon can't improve the cost much at all by perturbing the weighted input z^l_j. So far as the demon can tell, the neuron is already pretty near optimal[2]. And so there's a heuristic sense in which $\partial C/\partial z^l_j$ is a measure of the error in the neuron.

Motivated by this story, we define the error δ^l_j of neuron j in layer l by

$$\delta^l_j \equiv \frac{\partial C}{\partial z^l_j}. \tag{2.7}$$

As per our usual conventions, we use δl to denote the vector of errors associated with layer l. Backpropagation will give us a way of computing δl for every layer, and then relating those errors to the quantities of real interest, $\partial C/\partial w^l_{jk}$ and $\partial C/\partial b^l_j$.

You might wonder why the demon is changing the weighted input z^l_j. Surely it'd be more natural to imagine the demon changing the output activation a^l_j, with the result that we'd be using $\frac{\partial C}{\partial a^l_j}$ as our measure of error. In fact, if you do this things work out quite similarly to the discussion below. But it turns out to make the presentation of backpropagation a little more algebraically complicated. So we'll stick with $\delta^l_j = \frac{\partial C}{\partial z^l_j}$ as our measure of error[3].

Plan of attack: Backpropagation is based around four fundamental equations. Together, those equations give us a way of computing both the error δ^l and the gradient of the cost function. I state the four equations below. Be warned, though: you shouldn't expect to instantaneously assimilate the equations. Such an expectation will lead to disappointment. In fact, the backpropagation equations are so rich that understanding them well requires considerable time and patience as you gradually delve deeper into the equations. The good news is that such patience is repaid many times over. And so the discussion in this section is merely a beginning, helping you on the way to a thorough understanding of the equations.

Here's a preview of the ways we'll delve more deeply into the equations later in the chapter: I'll give a short proof of the equations, which helps explain why they are true; we'll restate the equations in algorithmic form as pseudocode, and see how the pseudocode can be implemented as real, running Python code; and, in the final section of the chapter, we'll develop an intuitive picture of what the backpropagation equations mean, and how someone might discover them from scratch. Along the way we'll return repeatedly to the four fundamental equations, and as you deepen your understanding those equations will come to seem comfortable and, perhaps, even beautiful and natural.

An equation for the error in the output layer, δ^L: The components of δ^L are given by

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j). \tag{BP1}$$

This is a very natural expression. The first term on the right, $\partial C/\partial a^L_j$, just measures how fast the cost is changing as a function of the j-th output activation. If, for example, C doesn't depend much on a particular output neuron, j, then δ^L_j will be small, which is what we'd expect. The second term on the right, $\sigma'(z^L_j)$, measures how fast the activation function σ is changing at z^L_j.

[2] This is only the case for small changes δz^l_j, of course. We'll assume that the demon is constrained to make such small changes.

[3] In classification problems like MNIST the term "error" is sometimes used to mean the classification failure rate. E.g., if the neural net correctly classifies 96.0 percent of the digits, then the error is 4.0 percent. Obviously, this has quite a different meaning from our δ vectors. In practice, you shouldn't have trouble telling which meaning is intended in any given usage.

Notice that everything in Eq. BP1 is easily computed. In particular, we compute z_j^L while computing the behaviour of the network, and it's only a small additional overhead to compute $\sigma'(z_j^L)$. The exact form of $\partial C/\partial a_j^L$ will, of course, depend on the form of the cost function. However, provided the cost function is known there should be little trouble computing $\partial C/\partial a_j^L$. For example, if we're using the quadratic cost function then $C = \frac{1}{2}\sum_j (y_j - a_j^L)^2$, and so $\partial C/\partial a_j^L = (a_j^L - y_j)$, which obviously is easily computable.

Equation BP1 is a componentwise expression for δ^L. It's a perfectly good expression, but not the matrix-based form we want for backpropagation. However, it's easy to rewrite the equation in a matrix-based form, as

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{BP1a}$$

Here, $\nabla_a C$ is defined to be a vector whose components are the partial derivatives $\partial C/\partial a_j^L$. You can think of $\nabla_a C$ as expressing the rate of change of C with respect to the output activations. It's easy to see that Equations BP1a and BP1 are equivalent, and for that reason from now on we'll use BP1 interchangeably to refer to both equations. As an example, in the case of the quadratic cost we have $\nabla_a C = (a^L - y)$, and so the fully matrix-based form of BP1 becomes

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \tag{30}$$

As you can see, everything in this expression has a nice vector form, and is easily computed using a library such as Numpy.

An equation for the error δ^l in terms of the error in the next layer, δ^{l+1}: In particular

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{BP2}$$

where $(w^{l+1})^T$ is the transpose of the weight matrix w^{l+1} for the $(l+1)$-th layer. This equation appears complicated, but each element has a nice interpretation. Suppose we know the error δ^{l+1} at the $(l+1)$-th layer. When we apply the transpose weight matrix, $(w^{l+1})^T$, we can think intuitively of this as moving the error *backward* through the network, giving us some sort of measure of the error at the output of the l-th layer. We then take the Hadamard product $\odot\sigma'(z^l)$. This moves the error backward through the activation function in layer l, giving us the error δ^l in the weighted input to layer l.

By combining (BP2) with (BP1) we can compute the error δ^l for any layer in the network. We start by using (BP1) to compute δ^L, then apply Equation (BP2) to compute δ^{L-1}, then Equation (BP2) again to compute δ^{L-2}, and so on, all the way back through the network.

An equation for the rate of change of the cost with respect to any bias in the network: In particular:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{BP3}$$

That is, the error δ_j^l is *exactly equal* to the rate of change $\partial C/\partial b_j^l$. This is great news, since (BP1) and (BP2) have already told us how to compute δ_j^l. We can rewrite (BP3) in shorthand as

$$\frac{\partial C}{\partial b} = \delta, \tag{2.8}$$

where it is understood that δ is being evaluated at the same neuron as the bias b.

**An equation for the rate of change of the cost with respect to any weight in the

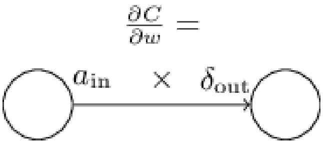
network: In particular:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j. \tag{BP4}$$

This tells us how to compute the partial derivatives $\partial C/\partial w^l_{jk}$ in terms of the quantities δ^l and a^{l-1}, which we already know how to compute. The equation can be rewritten in a less index-heavy notation as

$$\frac{\partial C}{\partial w} = a_{\rm in} \delta_{\rm out}, \tag{2.9}$$

where it's understood that $a_{\rm in}$ is the activation of the neuron input to the weight w, and $\delta_{\rm out}$ is the error of the neuron output from the weight w. Zooming in to look at just the weight w, and the two neurons connected by that weight, we can depict this as:



A nice consequence of Equation 2.9 is that when the activation $a_{\rm in}$ is small, $a_{\rm in} \approx 0$, the gradient term $\partial C/\partial w$ will also tend to be small. In this case, we'll say the weight *learns slowly*, meaning that it's not changing much during gradient descent. In other words, one consequence of (BP4) is that weights output from low-activation neurons learn slowly.

There are other insights along these lines which can be obtained from (BP1)–(BP4). Let's start by looking at the output layer. Consider the term $\sigma'(z^L_j)$ in (BP1). Recall from the graph of the sigmoid function in the last chapter that the σ function becomes very flat when $\sigma(z^L_j)$ is approximately 0 or 1. When this occurs we will have $\sigma'(z^L_j) \approx 0$. And so the lesson is that a weight in the final layer will learn slowly if the output neuron is either low activation (≈ 0) or high activation (≈ 1). In this case it's common to say the output neuron has *saturated* and, as a result, the weight has stopped learning (or is learning slowly). Similar remarks hold also for the biases of output neuron.

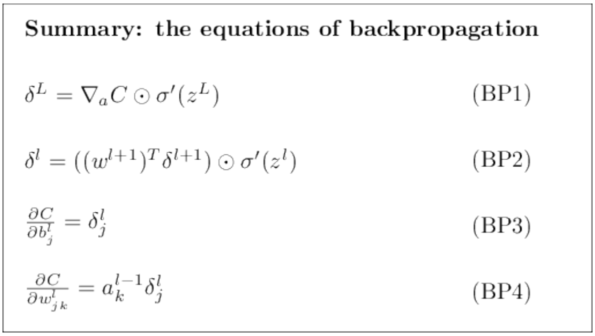
We can obtain similar insights for earlier layers. In particular, note the $\sigma'(z^l)$ term in (BP2). This means that δ^l_j is likely to get small if the neuron is near saturation. And this, in turn, means that any weights input to a saturated neuron will learn slowly[4].

Summing up, we've learnt that a weight will learn slowly if either the input neuron is low-activation, or if the output neuron has saturated, i.e., is either high- or low-activation.

None of these observations is too greatly surprising. Still, they help improve our mental model of what's going on as a neural network learns. Furthermore, we can turn this type of reasoning around. The four fundamental equations turn out to hold for any activation function, not just the standard sigmoid function (that's because, as we'll see in a moment, the proofs don't use any special properties of σ). And so we can use these equations to design activation functions which have particular desired learning properties. As an example to give you the idea, suppose we were to choose a (non-sigmoid) activation function σ so that σ' is always positive, and never gets close to zero. That would prevent the slow-down of learning that occurs when ordinary sigmoid neurons saturate. Later in the book we'll see examples where this kind of modification is made to the activation function. Keeping the

[4] This reasoning won't hold if $(w^{l+1})^T\delta^{l+1}$ has large enough entries to compensate for the smallness of $\sigma'(z^l_j)$. But I'm speaking of the general tendency.

four equations BP1–BP4 in mind can help explain why such modifications are tried, and what impact they can have.



Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \qquad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \qquad \text{(BP2)}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \qquad \text{(BP3)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \text{(BP4)}$$

Problem

- **Alternate presentation of the equations of backpropagation:** I've stated the equations of backpropagation (notably BP1 and BP2) using the Hadamard product. This presentation may be disconcerting if you're unused to the Hadamard product. There's an alternative approach, based on conventional matrix multiplication, which some readers may find enlightening.
  (1) Show that (BP1) may be rewritten as

$$\delta^L = \Sigma'(z^L)\nabla_a C, \tag{2.10}$$

  where $\sigma'(z^L)$ is a square matrix whose diagonal entries are the values $\sigma(z^L_j)$, and whose off-diagonal entries are zero. Note that this matrix acts on $\nabla_a C$ by conventional matrix multiplication.

  (2) Show that (BP2) may be rewritten as

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T \delta^{l+1}. \tag{2.11}$$

  (3) By combining observations (1) and (2) show that

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T \ldots \Sigma'(z^{L-1})(w^L)^T \Sigma'(z^L)\nabla_a C \tag{2.12}$$

  For readers comfortable with matrix multiplication this equation may be easier to understand than (BP1) and (BP2). The reason I've focused on (BP1) and (BP2) is because that approach turns out to be faster to implement numerically.

2.5 Proof of the four fundamental equations (optional)

We'll now prove the four fundamental equations (BP1)-(BP4). All four are consequences of the chain rule from multivariable calculus. If you're comfortable with the chain rule, then I strongly encourage you to attempt the derivation yourself before reading on.

Let's begin with Equation (BP1), which gives an expression for the output error, δ^l. To prove this equation, recall that by definition

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}. \tag{2.13}$$

Applying the chain rule, we can re-express the partial derivative above in terms of partial derivatives with respect to the output activations,

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}, \tag{2.14}$$

where the sum is over all neurons k in the output layer. Of course, the output activation a_k^L of the k-th neuron depends only on the weighted input z_j^L for the j-th neuron when $k = j$. And so $\partial a_k^L / \partial z_j^L$ vanishes when $k \neq j$. As a result we can simplify the previous equation to

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}. \tag{2.15}$$

Recalling that $a_j^L = \sigma(z_j^L)$ the second term on the right can be written as $\sigma'(z_j^L)$, and the equation becomes

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L), \tag{2.16}$$

which is just (BP1), in component form. Next, we'll prove (BP2), which gives an equation for the error δ^l in terms of the error in the next layer, δ^{l+1}. To do this, we want to rewrite $\delta_j^l = \partial C / \partial z_j^l$ in terms of $\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$. We can do this using the chain rule,

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}, \tag{2.17}$$

where in the last line we have interchanged the two terms on the right-hand side, and substituted the definition of δ_k^{l+1}. To evaluate the first term on the last line, note that

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}. \tag{2.18}$$

Differentiating, we obtain

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l). \tag{2.19}$$

Substituting back into (2.17) we obtain

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l). \tag{2.20}$$

This is just (BP2) written in component form.

The final two equations we want to prove are (BP3) and (BP4). These also follow from the chain rule, in a manner similar to the proofs of the two equations above. I leave them to you as an exercise.

Exercise

- Prove Equations (BP3) and (BP4).

That completes the proof of the four fundamental equations of backpropagation. The proof may seem complicated. But it's really just the outcome of carefully applying the chain rule. A little less succinctly, we can think of backpropagation as a way of computing the gradient of the cost function by systematically applying the chain rule from multi-variable calculus. That's all there really is to backpropagation – the rest is details.

2.6 The backpropagation algorithm

The backpropagation equations provide us with a way of computing the gradient of the cost function. Let's explicitly write this out in the form of an algorithm:

1. **Input** x: Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l al - 1 + b^l$ and $a^l = \sigma(z_l)$.
3. **Output error** δ^L: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Backpropagate the error**: For each $l = L—1, L—2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. **Output**: The gradient of the cost function is given by $\partial C \partial w^l_{jk} = a^{l-1}_k \delta^l_j$ and $\partial C \partial b^l_j = \delta^l_j$.

Examining the algorithm you can see why it's called *back*propagation. We compute the error vectors δ^l backward, starting from the final layer. It may seem peculiar that we're going through the network backward. But if you think about the proof of backpropagation, the backward movement is a consequence of the fact that the cost is a function of outputs from the network. To understand how the cost varies with earlier weights and biases we need to repeatedly apply the chain rule, working backward through the layers to obtain usable expressions.

Exercises

- **Backpropagation with a single modified neuron** Suppose we modify a single neuron in a feedforward network so that the output from the neuron is given by $f(\sum_j w_j x_j + b)$, where f is some function other than the sigmoid. How should we modify the backpropagation algorithm in this case?
- **Backpropagation with linear neurons** Suppose we replace the usual non-linear σ function with $\sigma(z) = z$ throughout the network. Rewrite the backpropagation algorithm for this case.

As I've described it above, the backpropagation algorithm computes the gradient of the cost function for a single training example, $C = C_x$. In practice, it's common to combine backpropagation with a learning algorithm such as stochastic gradient descent, in which we compute the gradient for many training examples. In particular, given a mini-batch of m training examples, the following algorithm applies a gradient descent learning step based on that mini-batch:

1. Input a set of training examples
2. For each training example x: Set the corresponding input activation $a^{x,1}$, and perform the following steps:
   - Feedforward: For each l= $2, 3, \ldots, L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$.
   - Output error $\delta^{x,L}$: Compute the vector $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.

   - Backpropagate the error: For each $l = L-1, L-2, \ldots, 2$ compute $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.

3. Gradient descent: For each $l = L, L-1, \ldots, 2$ update the weights according to the rule $w^l \rightarrow w^l - \frac{\eta}{m}\sum_x \delta^{x,l}(a^{x,l-1})^T$, and the biases according to the rule $b^l \rightarrow b^l - \frac{\eta}{m}\sum_x \delta^{x,l}$.

Of course, to implement stochastic gradient descent in practice you also need an outer loop generating mini-batches of training examples, and an outer loop stepping through multiple epochs of training. I've omitted those for simplicity.

2.7 The code for backpropagation

Having understood backpropagation in the abstract, we can now understand the code used in the last chapter to implement backpropagation. Recall from that chapter that the code was contained in the update_mini_batch and backprop methods of the Network class. The code for these methods is a direct translation of the algorithm described above. In particular, the update_mini_batch method updates the Network's weights and biases by computing the gradient for the current mini_batch of training examples:

```
class Network(object):
...
  def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch.
    The "mini_batch" is a list of tuples "(x, y)", and "eta"
    is the learning rate."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
      delta_nabla_b, delta_nabla_w = self.backprop(x, y)
      nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
      nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
            for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
            for b, nb in zip(self.biases, nabla_b)]
```

Most of the work is done by the line delta_nabla_b, delta_nabla_w = self.backprop(x, y) which uses the backprop method to figure out the partial derivatives $\partial C_x/\partial b^l_j$ and $\partial C_x/\partial w^l_{jk}$. The backprop method follows the algorithm in the last section closely. There is one small change - we use a slightly different approach to indexing the layers. This change is made to take advantage of a feature of Python, namely the use of negative list indices to count backward from the end of a list, so, e.g., l[-3] is the third last entry in a list l. The code for backprop is below, together with a few helper functions, which are used to compute the σ function, the derivative σ', and the derivative of the cost function. With these inclusions you should be able to understand the code in a self-contained way. If something's tripping you up, you may find it helpful to consult [link]the original description (and complete listing) of the code.

```
class Network(object):
...
  def backprop(self, x, y):
    """Return a tuple "(nabla_b, nabla_w)" representing the
    gradient for the cost function C_x.  "nabla_b" and
    "nabla_w" are layer-by-layer lists of numpy arrays, similar
    to "self.biases" and "self.weights"."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
      z = np.dot(w, activation)+b
      zs.append(z)
      activation = sigmoid(z)
      activations.append(activation)
    # backward pass
    delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    # Note that the variable l in the loop below is used a little
    # differently to the notation in Chapter 2 of the book. Here,
    # l = 1 means the last layer of neurons, l = 2 is the
    # second-last layer, and so on. It's a renumbering of the
    # scheme in the book, used here to take advantage of the fact
    # that Python can use negative indices in lists.
    for l in xrange(2, self.num_layers):
      z = zs[-l]
      sp = sigmoid_prime(z)
      delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
      nabla_b[-l] = delta
      nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
...
  def cost_derivative(self, output_activations, y):
    """Return the vector of partial derivatives \partial{} C_x /
    \partial{} a for the output activations."""
    return (output_activations-y)
  def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))
  def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

Problem

- **Fully matrix-based approach to backpropagation over a mini-batch** Our implementation of stochastic gradient descent loops over training examples in a mini-batch. It's possible to modify the backpropagation algorithm so that it computes the gradients for all training examples in a mini-batch simultaneously. The idea is that instead of beginning with a single input vector, x, we can begin with a matrix $X = [x_1 x_2 \ldots x_m]$ whose columns are the vectors in the mini-batch. We forward-propagate by multiplying by the weight matrices, adding a suitable matrix for the bias terms, and applying the sigmoid function everywhere. We backpropagate along similar lines. Explicitly write out pseudocode for this approach to the backpropagation algorithm. Modify network.py so that it uses this fully matrix-based approach. The advantage of this approach is that it takes full advantage of modern libraries for linear algebra. As a result it can be quite a bit faster than looping over the mini-batch. (On my laptop, for example, the speedup is about a factor of two when run on MNIST classification problems like those we considered in the last chapter.) In practice, all serious libraries for backpropagation use this fully matrix-based approach or some variant.

2.8 In what sense is backpropagation a fast algorithm?

In what sense is backpropagation a fast algorithm? To answer this question, let's consider another approach to computing the gradient. Imagine it's the early days of neural networks research. Maybe it's the 1950s or 1960s, and you're the first person in the world to think of using gradient descent to learn! But to make the idea work you need a way of computing the gradient of the cost function. You think back to your knowledge of calculus, and decide to see if you can use the chain rule to compute the gradient. But after playing around a bit, the algebra looks complicated, and you get discouraged. So you try to find another approach. You decide to regard the cost as a function of the weights $C = C(w)$ alone (we'll get back to the biases in a moment). You number the weights $w_1, w_2, \ldots$, and want to compute $\partial C/\partial w_j$ for some particular weight w_j. An obvious way of doing that is to use the approximation

$$\frac{\partial C}{\partial w_j} \approx \frac{C(w+\epsilon e_j)-C(w)}{\epsilon}, \tag{2.21}$$

where $\epsilon > 0$ is a small positive number, and e_j is the unit vector in the j-th direction. In other words, we can estimate $\partial C/\partial w_j$ by computing the cost C for two slightly different values of w_j, and then applying Equation 2.21. The same idea will let us compute the partial derivatives $\partial C/\partial b$ with respect to the biases.

This approach looks very promising. It's simple conceptually, and extremely easy to implement, using just a few lines of code. Certainly, it looks much more promising than the idea of using the chain rule to compute the gradient!

Unfortunately, while this approach appears promising, when you implement the code it turns out to be extremely slow. To understand why, imagine we have a million weights in our network. Then for each distinct weight w_j we need to compute $C(w + \epsilon e_j)$ in order to compute $\partial C/\partial w_j$. That means that to compute the gradient we need to compute the cost function a million different times, requiring a million forward passes through the network (per training example). We need to compute $C(w)$ as well, so that's a total of a million and one passes through the network.

What's clever about backpropagation is that it enables us to simultaneously compute all the partial derivatives $\partial C/\partial w_j$ using just one forward pass through the network, followed by one backward pass through the network. Roughly speaking, the computational cost of the backward pass is about the same as the forward pass[5]. And so the total cost of backpropagation is roughly the same as making just two forward passes through the network. Compare that to the million and one forward passes we needed for the approach based on (2.21)! And so even though backpropagation appears superficially more complex than the approach based on (2.21), it's actually much, much faster.

This speedup was first fully appreciated in 1986, and it greatly expanded the range of problems that neural networks could solve. That, in turn, caused a rush of people using neural networks. Of course, backpropagation is not a panacea. Even in the late 1980s people ran up against limits, especially when attempting to use backpropagation to train deep neural networks, i.e., networks with many hidden layers. Later in the book we'll see how modern computers and some clever new ideas now make it possible to use backpropagation to train such deep neural networks.

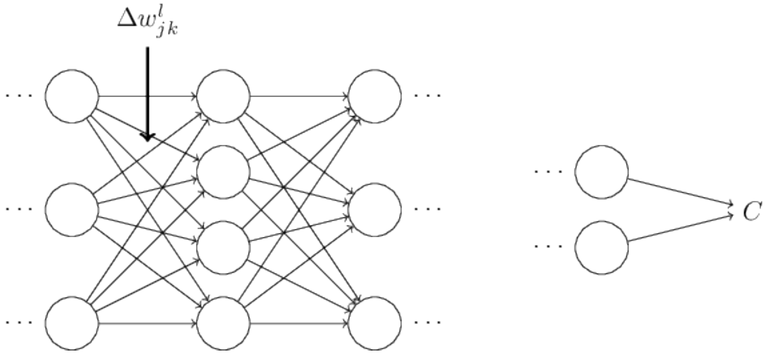
2.9 Backpropagation: the big picture

As I've explained it, backpropagation presents two mysteries. First, what's the algorithm really doing? We've developed a picture of the error being backpropagated from the output. But can we go any deeper, and build up more intuition about what is going on when we do all these matrix and vector multiplications? The second mystery is how someone could ever have discovered backpropagation in the first place? It's one thing to follow the steps in an algorithm, or even to follow the proof that the algorithm works. But that doesn't mean

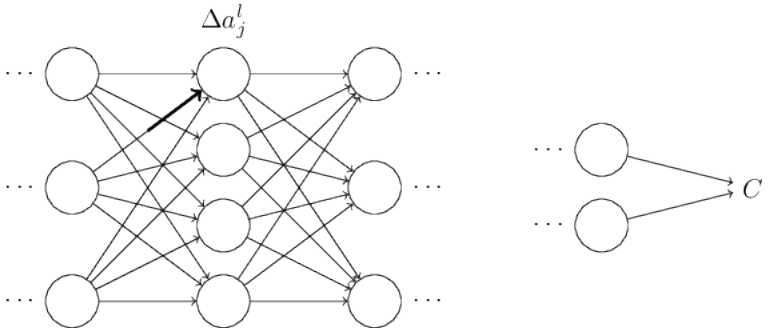
[5] This should be plausible, but it requires some analysis to make a careful statement. It's plausible because the dominant computational cost in the forward pass is multiplying by the weight matrices, while in the backward pass it's multiplying by the transposes of the weight matrices. These operations obviously have similar computational cost.

you understand the problem so well that you could have discovered the algorithm in the first place. Is there a plausible line of reasoning that could have led you to discover the backpropagation algorithm? In this section I'll address both these mysteries.

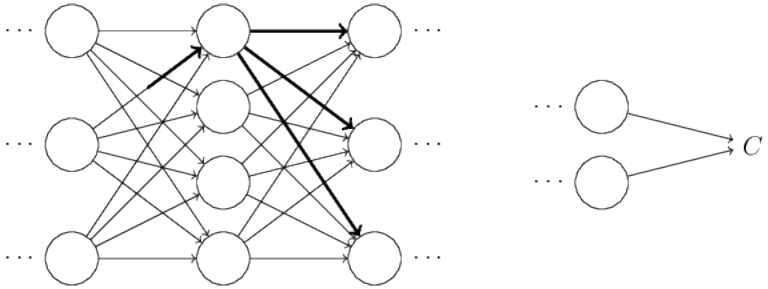
To improve our intuition about what the algorithm is doing, let's imagine that we've made a small change Δw^l_{jk} to some weight in the network, w^l_{jk}:



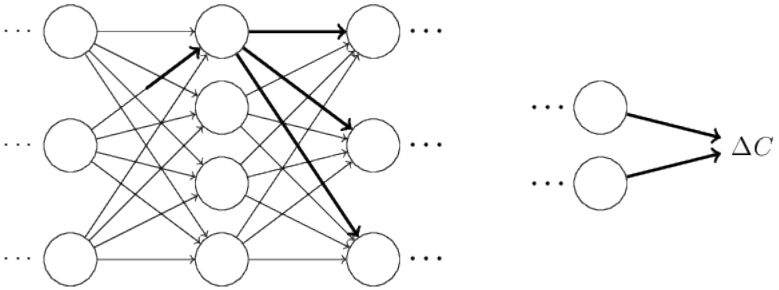
That change in weight will cause a change in the output activation from the corresponding neuron:



That, in turn, will cause a change in all the activations in the next layer:



Those changes will in turn cause changes in the next layer, and then the next, and so on all the way through to causing a change in the final layer, and then in the cost function:



The change ΔC in the cost is related to the change Δw^l_{jk} in the weight by the equation

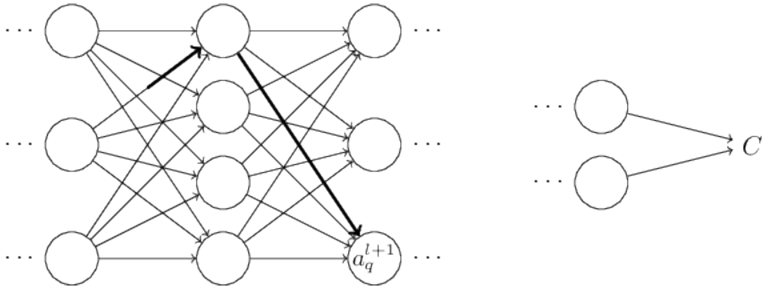
$$\Delta C \approx \frac{\partial C}{\partial w^l_{jk}} \Delta w^l_{jk}. \tag{2.22}$$

This suggests that a possible approach to computing $\partial C / \partial w^l_{jk}$ is to carefully track how a small change in w^l_{jk} propagates to cause a small change in C. If we can do that, being careful to express everything along the way in terms of easily computable quantities, then we should be able to compute $\partial C / \partial w^l_{jk}$.

Let's try to carry this out. The change Δw^l_{jk} causes a small change Δa^l_j in the activation of the j-th neuron in the l-th layer. This change is given by

$$\Delta a^l_j \approx \frac{\partial a^l_j}{\partial w^l_{jk}} \Delta w^l_{jk}. \tag{2.23}$$

The change in activation Δa^l_j will cause changes in all the activations in the next layer, i.e., the $(l+1)$-th layer. We'll concentrate on the way just a single one of those activations is affected, say a^{l+1}_q,



In fact, it'll cause the following change:

$$\Delta a^{l+1}_q \approx \frac{\partial a^{l+1}_q}{\partial a^l_j} \Delta a^l_j. \tag{2.24}$$

Substituting in the expression from Equation 2.23, we get:

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l. \tag{2.25}$$

Of course, the change Δa_q^{l+1} will, in turn, cause changes in the activations in the next layer. In fact, we can imagine a path all the way through the network from w_{jk}^l to C, with each change in activation causing a change in the next activation, and, finally, a change in the cost at the output. If the path goes through activations $a_j^l, a_q^{l+1}, \cdots, a_n^{L-1}, a_m^L$ then the resulting expression is

$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \dots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l, \tag{2.26}$$

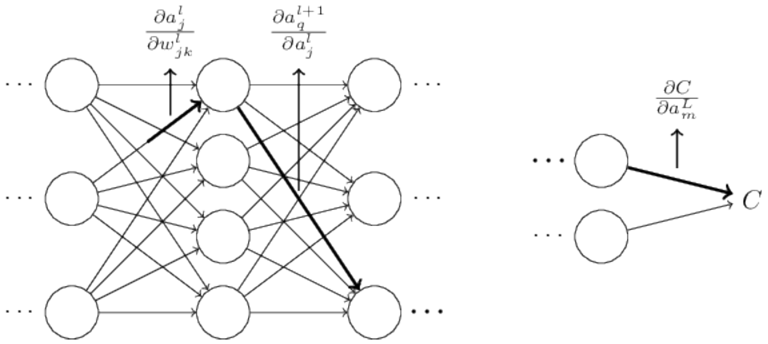
that is, we've picked up a $\partial a/\partial a$ type term for each additional neuron we've passed through, as well as the $\partial C/\partial a_m^L$ term at the end. This represents the change in C due to changes in the activations along this particular path through the network. Of course, there's many paths by which a change in w_{jk}^l can propagate to affect the cost, and we've been considering just a single path. To compute the total change in C it is plausible that we should sum over all the possible paths between the weight and the final cost, i.e.,

$$\Delta C \approx \sum_{mnp\dots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \dots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l, \tag{2.27}$$

where we've summed over all possible choices for the intermediate neurons along the path. Comparing with (2.22) we see that

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_{mnp\dots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \dots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l}. \tag{2.28}$$

Now, Equation 2.28 looks complicated. However, it has a nice intuitive interpretation. We're computing the rate of change of C with respect to a weight in the network. What the equation tells us is that every edge between two neurons in the network is associated with a rate factor which is just the partial derivative of one neuron's activation with respect to the other neuron's activation. The edge from the first weight to the first neuron has a rate factor $\partial a_j^l/\partial w_{jk}^l$. The rate factor for a path is just the product of the rate factors along the path. And the total rate of change $\partial C/\partial w_{jk}^l$ is just the sum of the rate factors of all paths from the initial weight to the final cost. This procedure is illustrated here, for a single path:


$\frac{\partial a^l_j}{\partial w^l_{jk}}$
$\frac{\partial a^{l+1}_q}{\partial a^l_j}$
$\frac{\partial C}{\partial a^L_m}$
C

What I've been providing up to now is a heuristic argument, a way of thinking about what's going on when you perturb a weight in a network. Let me sketch out a line of thinking you could use to further develop this argument. First, you could derive explicit expressions for all the individual partial derivatives in Equation 2.28. That's easy to do with a bit of calculus. Having done that, you could then try to figure out how to write all the sums over indices as matrix multiplications. This turns out to be tedious, and requires some persistence, but not extraordinary insight. After doing all this, and then simplifying as much as possible, what you discover is that you end up with exactly the backpropagation algorithm! And so you can think of the backpropagation algorithm as providing a way of computing the sum over the rate factor for all these paths. Or, to put it slightly differently, the backpropagation algorithm is a clever way of keeping track of small perturbations to the weights (and biases) as they propagate through the network, reach the output, and then affect the cost.

Now, I'm not going to work through all this here. It's messy and requires considerable care to work through all the details. If you're up for a challenge, you may enjoy attempting it. And even if not, I hope this line of thinking gives you some insight into what backpropagation is accomplishing.

What about the other mystery - how backpropagation could have been discovered in the first place? In fact, if you follow the approach I just sketched you will discover a proof of backpropagation. Unfortunately, the proof is quite a bit longer and more complicated than the one I described earlier in this chapter. So how was that short (but more mysterious) proof discovered? What you find when you write out all the details of the long proof is that, after the fact, there are several obvious simplifications staring you in the face. You make those simplifications, get a shorter proof, and write that out. And then several more obvious simplifications jump out at you. So you repeat again. The result after a few iterations is the proof we saw earlier[6] – short, but somewhat obscure, because all the signposts to its construction have been removed! I am, of course, asking you to trust me on this, but there really is no great mystery to the origin of the earlier proof. It's just a lot of hard work simplifying the proof I've sketched in this section.

[6] There is one clever step required. In Equation 2.28 the intermediate variables are activations like a_q^{l+1}. The clever idea is to switch to using weighted inputs, like z_q^{l+1}, as the intermediate variables. If you don't have this idea, and instead continue using the activations a_q^{l+1}, the proof you obtain turns out to be slightly more complex than the proof given earlier in the chapter.

3 Improving the way neural networks learn

When a golf player is first learning to play golf, they usually spend most of their time developing a basic swing. Only gradually do they develop other shots, learning to chip, draw and fade the ball, building on and modifying their basic swing. In a similar way, up to now we've focused on understanding the backpropagation algorithm. It's our "basic swing", the foundation for learning in most work on neural networks. In this chapter I explain a suite of techniques which can be used to improve on our vanilla implementation of backpropagation, and so improve the way our networks learn.

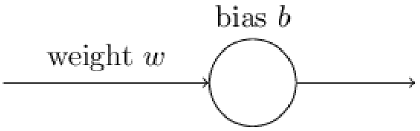
The techniques we'll develop in this chapter include: a better choice of cost function, known as the cross-entropy cost function; four so-called "regularization" methods (L1 and L2 regularization, dropout, and artificial expansion of the training data), which make our networks better at generalizing beyond the training data; a better method for initializing the weights in the network; and a set of heuristics to help choose good hyper-parameters for the network. I'll also overview several other techniques in less depth. The discussions are largely independent of one another, and so you may jump ahead if you wish. We'll also implement many of the techniques in running code, and use them to improve the results obtained on the handwriting classification problem studied in Chapter 1.

Of course, we're only covering a few of the many, many techniques which have been developed for use in neural nets. The philosophy is that the best entree to the plethora of available techniques is in-depth study of a few of the most important. Mastering those important techniques is not just useful in its own right, but will also deepen your understanding of what problems can arise when you use neural networks. That will leave you well prepared to quickly pick up other techniques, as you need them.

3.1 The cross-entropy cost function

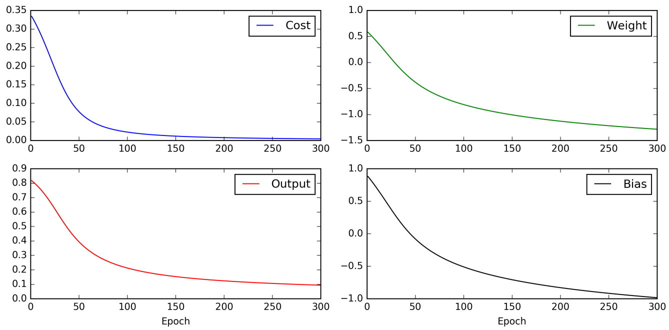
Most of us find it unpleasant to be wrong. Soon after beginning to learn the piano I gave my first performance before an audience. I was nervous, and began playing the piece an octave too low. I got confused, and couldn't continue until someone pointed out my error. I was very embarrassed. Yet while unpleasant, we also learn quickly when we're decisively wrong. You can bet that the next time I played before an audience I played in the correct octave! By contrast, we learn more slowly when our errors are less well-defined.

Ideally, we hope and expect that our neural networks will learn fast from their errors. Is this what happens in practice? To answer this question, let's look at a toy example. The example involves a neuron with just one input:

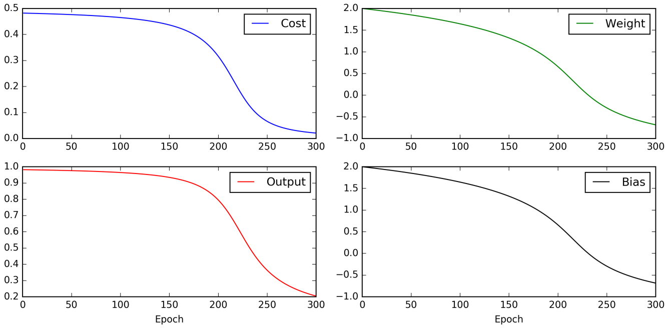


We'll train this neuron to do something ridiculously easy: take the input 1 to the output 0. Of course, this is such a trivial task that we could easily figure out an appropriate weight and bias by hand, without using a learning algorithm. However, it turns out to be illuminating to use gradient descent to attempt to learn a weight and bias. So let's take a look at how the neuron learns.

To make things definite, I'll pick the initial weight to be 0.6 and the initial bias to be 0.9. These are generic choices used as a place to begin learning, I wasn't picking them to be special in any way. The initial output from the neuron is 0.82, so quite a bit of learning will be needed before our neuron gets near the desired output, 0.0. The learning rate is η=0.15, which turns out to be slow enough that we can follow what's happening, but fast enough that we can get substantial learning in just a few seconds. The cost is the quadratic cost function, C, introduced back in Chapter 1. I'll remind you of the exact form of the cost function shortly, so there's no need to go and dig up the definition.



As you can see, the neuron rapidly learns a weight and bias that drives down the cost, and gives an output from the neuron of about 0.09. That's not quite the desired output, 0.0, but it is pretty good. Suppose, however, that we instead choose both the starting weight and the starting bias to be 2.0. In this case the initial output is 0.98, which is very badly wrong. Let's look at how the neuron learns to output 0 in this case.



Although this example uses the same learning rate ($\eta = 0.15$), we can see that learning starts out much more slowly. Indeed, for the first 150 or so learning epochs, the weights and biases don't change much at all. Then the learning kicks in and, much as in our first example, the neuron's output rapidly moves closer to 0.0.

This behavior is strange when contrasted to human learning. As I said at the beginning of this section, we often learn fastest when we're badly wrong about something. But we've just seen that our artificial neuron has a lot of difficulty learning when it's badly wrong – far more difficulty than when it's just a little wrong. What's more, it turns out that this behavior occurs not just in this toy model, but in more general networks. Why is learning so slow? And can we find a way of avoiding this slowdown?

To understand the origin of the problem, consider that our neuron learns by changing the weight and bias at a rate determined by the partial derivatives of the cost function, $\partial C/\partial w$ and $\partial C/\partial b$. So saying "learning is slow" is really the same as saying that those partial derivatives are small. The challenge is to understand why they are small. To understand that, let's compute the partial derivatives. Recall that we're using the quadratic cost function, which, from Equation 1.6, is given by

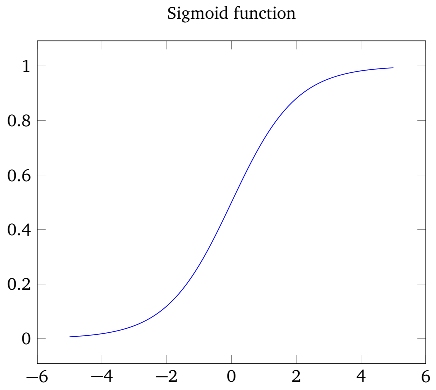
$$C = \frac{(y-a)^2}{2}, \tag{3.1}$$

where a is the neuron's output when the training input $x = 1$ is used, and $y = 0$ is the corresponding desired output. To write this more explicitly in terms of the weight and bias, recall that $a = \sigma(z)$, where $z = wx + b$. Using the chain rule to differentiate with respect to the weight and bias we get

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z) \tag{3.2}$$

$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z), \tag{3.3}$$

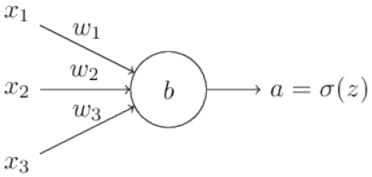
where I have substituted $x = 1$ and $y = 0$. To understand the behaviour of these expressions, let's look more closely at the $\sigma'(z)$ term on the right-hand side. Recall the shape of the σ function:



We can see from this graph that when the neuron's output is close to 1, the curve gets very flat, and so $\sigma'(z)$ gets very small. Equations 3.2 and 3.3 then tell us that $\partial C/\partial w$ and $\partial C/\partial b$ get very small. This is the origin of the learning slowdown. What's more, as we shall see a little later, the learning slowdown occurs for essentially the same reason in more general neural networks, not just the toy example we've been playing with.

3.1.1 Introducing the cross-entropy cost function

How can we address the learning slowdown? It turns out that we can solve the problem by replacing the quadratic cost with a different cost function, known as the cross-entropy. To understand the cross-entropy, let's move a little away from our super-simple toy model. We'll suppose instead that we're trying to train a neuron with several input variables, $x_1, x_2, \ldots$, corresponding weights $w_1, w_2, \ldots$, and a bias, b:



The output from the neuron is, of course, $a = \sigma(z)$, where $z = \sum_j w_j b_j + b$ is the weighted sum of the inputs. We define the cross-entropy cost function for this neuron by

$$C = -\frac{1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)], \tag{3.4}$$

where n is the total number of items of training data, the sum is over all training inputs, x, and y is the corresponding desired output.

It's not obvious that the expression (3.4) fixes the learning slowdown problem. In fact, frankly, it's not even obvious that it makes sense to call this a cost function! Before addressing the learning slowdown, let's see in what sense the cross-entropy can be interpreted as a cost function.

Two properties in particular make it reasonable to interpret the cross-entropy as a cost function. First, it's non-negative, that is, $C > 0$. To see this, notice that: (a) all the individual terms in the sum in (3.4) are negative, since both logarithms are of numbers in the range 0 to 1; and (b) there is a minus sign out the front of the sum.

Second, if the neuron's actual output is close to the desired output for all training inputs, x, then the cross-entropy will be close to zero[1]. To see this, suppose for example that y=0 and aâĽĹ0 for some input x. This is a case when the neuron is doing a good job on that input. We see that the first term in the expression (57) for the cost vanishes, since $y = 0$, while the second term is just $-\ln(1-a) \approx 0$. A similar analysis holds when $y = 1$ and $a \approx 1$. And so the contribution to the cost will be low provided the actual output is close to the desired output.

Summing up, the cross-entropy is positive, and tends toward zero as the neuron gets better at computing the desired output, y, for all training inputs, x. These are both properties we'd intuitively expect for a cost function. Indeed, both properties are also satisfied by the quadratic cost. So that's good news for the cross-entropy. But the cross-entropy cost function has the benefit that, unlike the quadratic cost, it avoids the problem of learning slowing down. To see this, let's compute the partial derivative of the cross-entropy cost with respect to the weights. We substitute $a = \sigma(z)$ into (3.4), and apply the chain rule twice, obtaining:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial w_j} = -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\sigma'(z)x_j. \tag{3.5}$$

Putting everything over a common denominator and simplifying this becomes:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z)-y). \tag{3.6}$$

Using the definition of the sigmoid function, $\sigma(z) = 1/(1+e^{-z})$, and a little algebra we can show that $\sigma'(z) = \sigma(z)(1-\sigma(z))$. I'll ask you to verify this in an exercise below, but for now let's accept it as given. We see that the $\sigma'(z)$ and $\sigma(z)(1-\sigma(z))$ terms cancel in the equation just above, and it simplifies to become:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z)-y). \tag{3.7}$$

This is a beautiful expression. It tells us that the rate at which the weight learns is controlled by $\sigma(z)-y$, i.e., by the error in the output. The larger the error, the faster the neuron will learn. This is just what we'd intuitively expect. In particular, it avoids the learning slowdown caused by the $\sigma'(z)$ term in the analogous equation for the quadratic cost, Equation(3.2). When we use the cross-entropy, the $\sigma'(z)$ term gets canceled out, and we no longer need worry about it being small. This cancellation is the special miracle ensured by the cross-entropy cost function. Actually, it's not really a miracle. As we'll see later, the cross-entropy was specially chosen to have just this property.

In a similar way, we can compute the partial derivative for the bias. I won't go through all the details again, but you can easily verify that

$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x (\sigma(z)-y). \tag{3.8}$$

Again, this avoids the learning slowdown caused by the $\sigma'(z)$ term in the analogous equation

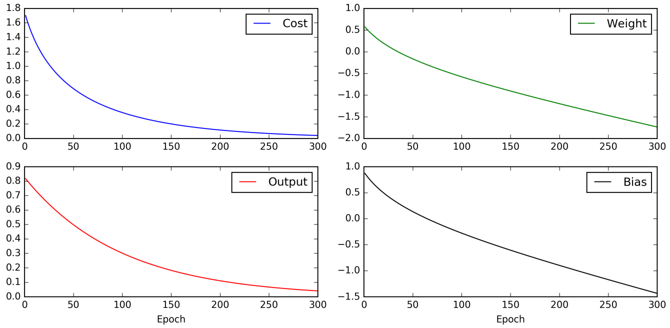
[1] To prove this I will need to assume that the desired outputs y are all either 0 or 1. This is usually the case when solving classification problems, for example, or when computing Boolean functions. To understand what happens when we don't make this assumption, see the exercises at the end of this section.

for the quadratic cost, Equation (3.3).

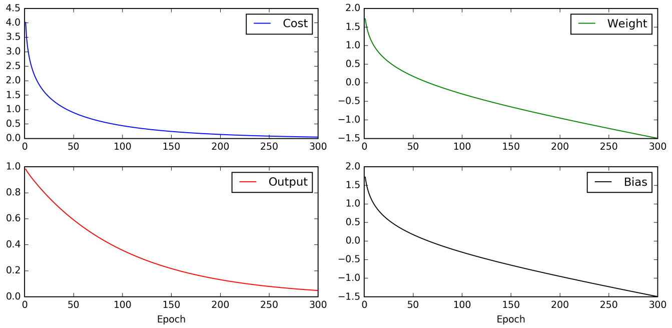
Exercise

- Verify that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$

Let's return to the toy example we played with earlier, and explore what happens when we use the cross-entropy instead of the quadratic cost. To re-orient ourselves, we'll begin with the case where the quadratic cost did just fine, with starting weight 0.6 and starting bias 0.9 :



Unsurprisingly, the neuron learns perfectly well in this instance, just as it did earlier. And now let's look at the case where our neuron got stuck before ([link], for comparison), with the weight and bias both starting at 2.0:



Success! This time the neuron learned quickly, just as we hoped. If you observe closely you can see that the slope of the cost curve was much steeper initially than the initial flat region on the corresponding curve for the quadratic cost. It's that steepness which the cross-entropy buys us, preventing us from getting stuck just when we'd expect our neuron to learn fastest, i.e., when the neuron starts out badly wrong.

I didn't say what learning rate was used in the examples just illustrated. Earlier, with the quadratic cost, we used $\eta = 0.15$. Should we have used the same learning rate in the new examples? In fact, with the change in cost function it's not possible to say precisely what it means to use the "same" learning rate; it's an apples and oranges comparison. For both cost functions I simply experimented to find a learning rate that made it possible to see what is going on. If you're still curious, despite my disavowal, here's the lowdown: I used $\eta = 0.005$ in the examples just given.

You might object that the change in learning rate makes the graphs above meaningless. Who cares how fast the neuron learns, when our choice of learning rate was arbitrary to begin with?! That objection misses the point. The point of the graphs isn't about the absolute speed of learning. It's about how the speed of learning changes. In particular, when we use the quadratic cost learning is slower when the neuron is unambiguously wrong than it is later on, as the neuron gets closer to the correct output; while with the cross-entropy learning is faster when the neuron is unambiguously wrong. Those statements don't depend on how the learning rate is set.

We've been studying the cross-entropy for a single neuron. However, it's easy to generalize the cross-entropy to many-neuron multi-layer networks. In particular, suppose $y = y_1, y_2, \ldots$ are the desired values at the output neurons, i.e., the neurons in the final layer, while $a^L_1, a^L_2, \ldots$ are the actual output values. Then we define the cross-entropy by

$$\sum_j \left[y_j \ln a^L_j + (1 - y_j) \ln(1 - a^L_j) \right]. \tag{3.9}$$

This is the same as our earlier expression, Equation (3.4), except now we've got the $\sum_j$ summing over all the output neurons. I won't explicitly work through a derivation, but it should be plausible that using the expression (3.9) avoids a learning slowdown in many-neuron networks. If you're interested, you can work through the derivation in the problem below.

Incidentally, I'm using the term "cross-entropy" in a way that has confused some early readers, since it superficially appears to conflict with other sources. In particular, it's common to define the cross-entropy for two probability distributions, p_j and q_j, as $\sum_j p_j \ln q_j$. This definition may be connected to (3.4), if we treat a single sigmoid neuron as outputting a probability distribution consisting of the neuron's activation a and its complement $1 - a$.

However, when we have many sigmoid neurons in the final layer, the vector a^L_j of activations don't usually form a probability distribution. As a result, a definition like $\sum_j p_j \ln q_j$ doesn't even make sense, since we're not working with probability distributions. Instead, you can think of (3.9) as a summed set of per-neuron cross-entropies, with the activation of each neuron being interpreted as part of a two-element probability distribution[2]. In this sense, (3.9) is a generalization of the cross-entropy for probability distributions.

When should we use the cross-entropy instead of the quadratic cost? In fact, the cross-entropy is nearly always the better choice, provided the output neurons are sigmoid neurons. To see why, consider that when we're setting up the network we usually initialize the weights and biases using some sort of randomization. It may happen that those initial choices result in the network being decisively wrong for some training input - that is, an output neuron will have saturated near 1, when it should be 0, or vice versa. If we're using the quadratic cost that will slow down learning. It won't stop learning completely, since the weights will continue learning from other training inputs, but it's obviously undesirable.

Exercises

- One gotcha with the cross-entropy is that it can be difficult at first to remember the respective roles of the ys and the as. It's easy to get confused about whether the right form is

$$-[y \ln a + (1 - y) \ln(1 - a)].$$

  What happens to the second of these expressions when $y = 0$ or 1? Does this problem afflict the first expression? Why or why not?
- In the single-neuron discussion at the start of this section, I argued that the cross-entropy is small if $\sigma(z) \approx y$ for all training inputs. The argument relied on y

[2] Of course, in our networks there are no probabilistic elements, so they're not really probabilities.

being equal to either 0 or 1. This is usually true in classification problems, but for other problems (e.g., regression problems) y can sometimes take values intermediate between 0 and 1. Show that the cross-entropy is still minimized when $\sigma(z) = y$ for all training inputs. When this is the case the cross-entropy has the value:

$$C = -\frac{1}{n} \sum_x [y \ln y + (1 - y) \ln(1 - y)]. \tag{3.10}$$

The quantity $-[y \ln y + (1 - y) \ln(1 - y)]$ is sometimes known as the *binary entropy*.

Problems

- **Many-layer multi-neuron networks** In the notation introduced in the last chapter, show that for the quadratic cost the partial derivative with respect to weights in the output layer is
$$\frac{\partial C}{\partial w^L_{jk}} = \frac{1}{n} \sum_x a^{L-1}_k (a^L_j - y_j) \sigma'(z^L_j). \tag{3.11}$$
The term $\sigma'(z^L_j)$ causes a learning slowdown whenever an output neuron saturates on the wrong value. Show that for the cross-entropy cost the output error δ^L for a single training example x is given by
$$\delta^L = a^L - y. \tag{3.12}$$
Use this expression to show that the partial derivative with respect to the weights in the output layer is given by
$$\frac{\partial C}{\partial w^L_{jk}} = \frac{1}{n} \sum_x a^{L-1}_k (a^L_j - y_j). \tag{3.13}$$
The $\sigma'(z^L_j)$ term has vanished, and so the cross-entropy avoids the problem of learning slowdown, not just when used with a single neuron, as we saw earlier, but also in many-layer multi-neuron networks. A simple variation on this analysis holds also for the biases. If this is not obvious to you, then you should work through that analysis as well.
- **Using the quadratic cost when we have linear neurons in the output layer** Suppose that we have a many-layer multi-neuron network. Suppose all the neurons in the final layer are linear neurons, meaning that the sigmoid activation function is not applied, and the outputs are simply $a^L_j = z^L_j$. Show that if we use the quadratic cost function then the output error δ^L for a single training example x is given by
$$\delta^L = a^L - y. \tag{3.14}$$
Similarly to the previous problem, use this expression to show that the partial derivatives with respect to the weights and biases in the output layer are given by
$$\begin{aligned} \frac{\partial C}{\partial w^L_{jk}} &= \frac{1}{n} \sum_x a^{L-1}_k (a^L_j - y_j) && (3.15) \\ \frac{\partial C}{\partial b^L_j} &= \frac{1}{n} \sum_x (a^L_j - y_j). && (3.16) \end{aligned}$$

This shows that if the output neurons are linear neurons then the quadratic cost will not give rise to any problems with a learning slowdown. In this case the quadratic cost is, in fact, an appropriate cost function to use.

3.1.2 Using the cross-entropy to classify MNIST digits

The cross-entropy is easy to implement as part of a program which learns using gradient descent and backpropagation. We'll do that later in the chapter, developing an improved version of our earlier program for classifying the MNIST handwritten digits, `network.py`. The new program is called `network2.py`, and incorporates not just the cross-entropy, but also several other techniques developed in this chapter[3]. For now, let's look at how well our new program classifies MNIST digits. As was the case in Chapter 1, we'll use a network with 30 hidden neurons, and we'll use a mini-batch size of 10. We set the learning rate to $\eta = 0.5$[4] and we train for 30 epochs. The interface to `network2.py` is slightly different than `network.py`, but it should still be clear what is going on. You can, by the way, get documentation about `network2.py`'s interface by using commands such as `help(network2.Network.SGD)` in a Python shell.

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data,
      monitor_evaluation_accuracy=True)
```

Note, by the way, that the `net.large_weight_initializer()` command is used to initialize the weights and biases in the same way as described in Chapter 1. We need to run this command because later in this chapter we'll change the default weight initialization in our networks. The result from running the above sequence of commands is a network with 95.49 percent accuracy. This is pretty close to the result we obtained in Chapter 1, 95.42 percent, using the quadratic cost.

Let's look also at the case where we use 100 hidden neurons, the cross-entropy, and otherwise keep the parameters the same. In this case we obtain an accuracy of 96.82 percent. That's a substantial improvement over the results from Chapter 1, where we obtained a classification accuracy of 96.59 percent, using the quadratic cost. That may look like a small change, but consider that the error rate has dropped from 3.41 percent to 3.18 percent. That is, we've eliminated about one in fourteen of the original errors. That's quite a handy improvement.

It's encouraging that the cross-entropy cost gives us similar or better results than the quadratic cost. However, these results don't conclusively prove that the cross-entropy is a better choice. The reason is that I've put only a little effort into choosing hyper-parameters such as learning rate, mini-batch size, and so on. For the improvement to be really convincing we'd need to do a thorough job optimizing such hyper-parameters. Still, the results are

[3] The code is available on [link]GitHub.

[4] In Chapter 1 we used the quadratic cost and a learning rate of $\eta = 3.0$. As discussed above, it's not possible to say precisely what it means to use the "same" learning rate when the cost function is changed. For both cost functions I experimented to find a learning rate that provides near-optimal performance, given the other hyper-parameter choices. There is, incidentally, a very rough general heuristic for relating the learning rate for the cross-entropy and the quadratic cost. As we saw earlier, the gradient terms for the quadratic cost have an extra $\sigma' = \sigma(1-\sigma)$ term in them. Suppose we average this over values for σ, $\int_0^1 d\sigma \sigma(1-\sigma) = 1/6$. We see that (very roughly) the quadratic cost learns an average of 6 times slower, for the same learning rate. This suggests that a reasonable starting point is to divide the learning rate for the quadratic cost by 6. Of course, this argument is far from rigorous, and shouldn't be taken too seriously. Still, it can sometimes be a useful starting point.

encouraging, and reinforce our earlier theoretical argument that the cross-entropy is a better choice than the quadratic cost.

This, by the way, is part of a general pattern that we'll see through this chapter and, indeed, through much of the rest of the book. We'll develop a new technique, we'll try it out, and we'll get "improved" results. It is, of course, nice that we see such improvements. But the interpretation of such improvements is always problematic. They're only truly convincing if we see an improvement after putting tremendous effort into optimizing all the other hyper-parameters. That's a great deal of work, requiring lots of computing power, and we're not usually going to do such an exhaustive investigation. Instead, we'll proceed on the basis of informal tests like those done above. Still, you should keep in mind that such tests fall short of definitive proof, and remain alert to signs that the arguments are breaking down.

By now, we've discussed the cross-entropy at great length. Why go to so much effort when it gives only a small improvement to our MNIST results? Later in the chapter we'll see other techniques – notably, regularization – which give much bigger improvements. So why so much focus on cross-entropy? Part of the reason is that the cross-entropy is a widely-used cost function, and so is worth understanding well. But the more important reason is that neuron saturation is an important problem in neural nets, a problem we'll return to repeatedly throughout the book. And so I've discussed the cross-entropy at length because it's a good laboratory to begin understanding neuron saturation and how it may be addressed.

3.1.3 What does the cross-entropy mean? Where does it come from?

Our discussion of the cross-entropy has focused on algebraic analysis and practical implementation. That's useful, but it leaves unanswered broader conceptual questions, like: what does the cross-entropy mean? Is there some intuitive way of thinking about the cross-entropy? And how could we have dreamed up the cross-entropy in the first place?

Let's begin with the last of these questions: what could have motivated us to think up the cross-entropy in the first place? Suppose we'd discovered the learning slowdown described earlier, and understood that the origin was the $\sigma'(z)$ terms in Equations (3.2) and (3.3). After staring at those equations for a bit, we might wonder if it's possible to choose a cost function so that the $\sigma'(z)$ term disappeared. In that case, the cost $C = C_x$ for a single training example x would satisfy

$$\frac{\partial C}{\partial w_j} = x_j(a-y) \tag{3.17}$$

$$\frac{\partial C}{\partial b} = (a-y). \tag{3.18}$$

If we could choose the cost function to make these equations true, then they would capture in a simple way the intuition that the greater the initial error, the faster the neuron learns. They'd also eliminate the problem of a learning slowdown. In fact, starting from these equations we'll now show that it's possible to derive the form of the cross-entropy, simply by following our mathematical noses. To see this, note that from the chain rule we have

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\sigma'(z). \tag{3.19}$$

Using $\sigma'(z) = \sigma(z)(1\sigma(z)) = a(1a)$ the last equation becomes

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a} a(1-a). \tag{3.20}$$

Comparing to Equation 3.18 we obtain

$$\frac{\partial C}{\partial a} = \frac{a-y}{a(1-a)}. \tag{3.21}$$

Integrating this expression with respect to a gives

$$C = -[y \ln a + (1-y)\ln(1-a)] + \text{constant}, \tag{3.22}$$

for some constant of integration. This is the contribution to the cost from a single training example, x. To get the full cost function we must average over training examples, obtaining

$$C = -\frac{1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)] + \text{constant}, \tag{3.23}$$

where the constant here is the average of the individual constants for each training example. And so we see that Equations (3.17) and (3.18) uniquely determine the form of the cross-entropy, up to an overall constant term. The cross-entropy isn't something that was miraculously pulled out of thin air. Rather, it's something that we could have discovered in a simple and natural way.

What about the intuitive meaning of the cross-entropy? How should we think about it? Explaining this in depth would take us further afield than I want to go. However, it is worth mentioning that there is a standard way of interpreting the cross-entropy that comes from the field of information theory. Roughly speaking, the idea is that the cross-entropy is a measure of surprise. In particular, our neuron is trying to compute the function $x \to y = y(x)$. But instead it computes the function $x \to a = a(x)$. Suppose we think of a as our neuron's estimated probability that y is 1, and $1-a$ is the estimated probability that the right value for y is 0. Then the cross-entropy measures how "surprised" we are, on average, when we learn the true value for y. We get low surprise if the output is what we expect, and high surprise if the output is unexpected. Of course, I haven't said exactly what "surprise" means, and so this perhaps seems like empty verbiage. But in fact there is a precise information-theoretic way of saying what is meant by surprise. Unfortunately, I don't know of a good, short, self-contained discussion of this subject that's available online. But if you want to dig deeper, then Wikipedia contains a [link]brief summary that will get you started down the right track. And the details can be filled in by working through the materials about the Kraft inequality in chapter 5 of the book about information theory by [link]Cover and Thomas.

Problem

- We've discussed at length the learning slowdown that can occur when output neurons saturate, in networks using the quadratic cost to train. Another factor that may inhibit learning is the presence of the x_j term in Equation (3.7). Because of this term, when an input x_j is near to zero, the corresponding weight w_j will learn slowly. Explain why it is not possible to eliminate the x_j term through a clever choice of cost function.

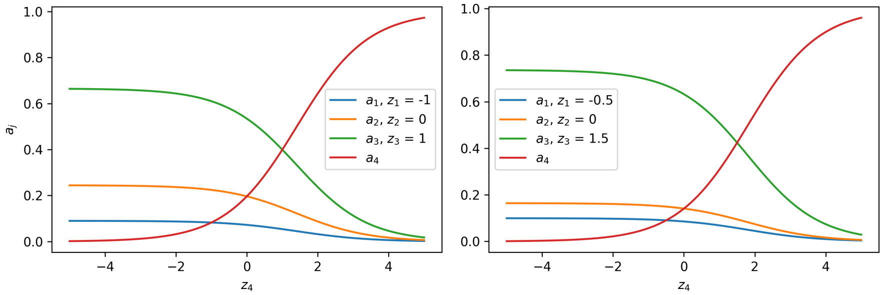


Figure 3.1: Equation 3.24 for different fixed values of $z^L_{1,2,3}$ and variable z^L_4. L index avoided for clarity.

3.1.4 Softmax

In this chapter we'll mostly use the cross-entropy cost to address the problem of learning slowdown. However, I want to briefly describe another approach to the problem, based on what are called *softmax* layers of neurons. We're not actually going to use softmax layers in the remainder of the chapter, so if you're in a great hurry, you can skip to the next section. However, softmax is still worth understanding, in part because it's intrinsically interesting, and in part because we'll use softmax layers in Chapter 6, in our discussion of deep neural networks.

The idea of softmax is to define a new type of output layer for our neural networks. It begins in the same way as with a sigmoid layer, by forming the weighted inputs[5] $z^L_j = \sum_k w^L_{jk} a^{L1}_k + b^L_j$. However, we don't apply the sigmoid function to get the output. Instead, in a softmax layer we apply the so-called softmax function to the z^L_j. According to this function, the activation a^L_j of the j-th output neuron is

$$a^L_j = \frac{e^{x^L_j}}{\sum_k e^{z^L_k}} \tag{3.24}$$

where in the denominator we sum over all the output neurons.

If you're not familiar with the softmax function, Equation (3.24) may look pretty opaque. It's certainly not obvious why we'd want to use this function. And it's also not obvious that this will help us address the learning slowdown problem. To better understand Equation (3.24), suppose we have a network with four output neurons, and four corresponding weighted inputs, which we'll denote z^L_1, z^L_2, z^L_3, and z^L_4. Figure3.1 shows a graph of the corresponding output activations for different inputs[6].

As you increase z^L_4, you'll see an increase in the corresponding output activation, a^L_4, and a decrease in the other output activations. Similarly, if you decrease z^L_4 then a^L_4 will decrease, and all the other output activations will increase. In fact, if you look closely, you'll see that in both cases the total change in the other activations exactly compensates for the change in a^L_4. The reason is that the output activations are guaranteed to always sum up to 1, as we

[5] In describing the softmax we'll make frequent use of notation introduced in the last chapter. You may wish to revisit that chapter if you need to refresh your memory about the meaning of the notation.

[6] This paragraph is an adaptation of an animation from online version of the book.

can prove using Equation (3.24) and a little algebra:

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1. \tag{3.25}$$

As a result, if a_4^L increases, then the other output activations must decrease by the same total amount, to ensure the sum over all activations remains 1. And, of course, similar statements hold for all the other activations.

Equation (3.24) also implies that the output activations are all positive, since the exponential function is positive. Combining this with the observation in the last paragraph, we see that the output from the softmax layer is a set of positive numbers which sum up to 1. In other words, the output from the softmax layer can be thought of as a probability distribution.

The fact that a softmax layer outputs a probability distribution is rather pleasing. In many problems it's convenient to be able to interpret the output activation a_j^L as the network's estimate of the probability that the correct output is j. So, for instance, in the MNIST classification problem, we can interpret a_j^L as the network's estimated probability that the correct digit classification is j.

By contrast, if the output layer was a sigmoid layer, then we certainly couldn't assume that the activations formed a probability distribution. I won't explicitly prove it, but it should be plausible that the activations from a sigmoid layer won't in general form a probability distribution. And so with a sigmoid output layer we don't have such a simple interpretation of the output activations.

Exercise

- Construct an example showing explicitly that in a network with a sigmoid output layer, the output activations a_j^L won't always sum to 1.

We're starting to build up some feel for the softmax function and the way softmax layers behave. Just to review where we're at: the exponentials in Equation (3.24) ensure that all the output activations are positive. And the sum in the denominator of Equation (3.24) ensures that the softmax outputs sum to 1. So that particular form no longer appears so mysterious: rather, it is a natural way to ensure that the output activations form a probability distribution. You can think of softmax as a way of rescaling the z_j^L, and then squishing them together to form a probability distribution.

Exercises

- **Monotonicity of softmax** Show that $\partial a_j^L / \partial z_k^L$is positive if $j = k$ and negative if $j \neq k$. As a consequence, increasing z_j^L is guaranteed to increase the corresponding output activation, a_j^L, and will decrease all the other output activations. We already saw this empirically with the sliders, but this is a rigorous proof.
- **Non-locality of softmax** A nice thing about sigmoid layers is that the output a_j^L is a function of the corresponding weighted input, $a_j^L = \sigma(z_j^L)$. Explain why this is not the case for a softmax layer: any particular output activation a_j^L depends on all the weighted inputs.

Problem

- **Inverting the softmax layer** Suppose we have a neural network with a softmax output layer, and the activations a_j^L are known. Show that the corresponding weighted inputs have the form $z_j^L = \ln a_j^L + C$, for some constant C that is independent of j.

The learning slowdown problem: We've now built up considerable familiarity with softmax layers of neurons. But we haven't yet seen how a softmax layer lets us address the learning slowdown problem. To understand that, let's define the *log-likelihood* cost function. We'll use x to denote a training input to the network, and y to denote the corresponding desired output. Then the log-likelihood cost associated to this training input is

$$C \equiv -\ln a_j^L \tag{3.26}$$

So, for instance, if we're training with MNIST images, and input an image of a 7, then the log-likelihood cost is $-\ln a_7^L$. To see that this makes intuitive sense, consider the case when the network is doing a good job, that is, it is confident the input is a 7. In that case it will estimate a value for the corresponding probability a_7^L which is close to 1, and so the cost $-\ln a_7^L$ will be small. By contrast, when the network isn't doing such a good job, the probability a_7^L will be smaller, and the cost $-\ln a_7^L$ will be larger. So the log-likelihood cost behaves as we'd expect a cost function to behave.

What about the learning slowdown problem? To analyze that, recall that the key to the learning slowdown is the behaviour of the quantities $\partial C/\partial w_{jk}^L$ and $\partial C/\partial b_j^L$. I won't go through the derivation explicitly – I'll ask you to do in the problems, below – but with a little algebra you can show that[7]

$$\frac{\partial C}{\partial b_j^L} = a_j^L - y_j \tag{3.27}$$

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}(a_j^L - y_j) \tag{3.28}$$

These equations are the same as the analogous expressions obtained in our earlier analysis of the cross-entropy. Compare, for example, Equation (3.28) to Equation (3.13). It's the same equation, albeit in the latter I've averaged over training instances. And, just as in the earlier analysis, these expressions ensure that we will not encounter a learning slowdown. In fact, it's useful to think of a softmax output layer with log-likelihood cost as being quite similar to a sigmoid output layer with cross-entropy cost.

Given this similarity, should you use a sigmoid output layer and cross-entropy, or a softmax output layer and log-likelihood? In fact, in many situations both approaches work well. Through the remainder of this chapter we'll use a sigmoid output layer, with the cross-entropy cost. Later, in Chapter 6, we'll sometimes use a softmax output layer, with log-likelihood cost. The reason for the switch is to make some of our later networks more similar to networks found in certain influential academic papers. As a more general point of principle, softmax plus log-likelihood is worth using whenever you want to interpret the output activations as probabilities. That's not always a concern, but can be useful with classification problems (like MNIST) involving disjoint classes.

Problems

- Derive Equations (3.27) and (3.28).
- **Where does the "softmax" name come from?** Suppose we change the softmax

[7] Note that I'm abusing notation here, using y in a slightly different way to last paragraph. In the last paragraph we used y to denote the desired output from the network – e.g., output a "7" if an image of a 7 was input. But in the equations which follow I'm using y to denote the vector of output activations which corresponds to 7, that is, a vector which is all 0s, except for a 1 in the 7th location.

function so the output activations are given by

$$a_j^L = \frac{e^{cz_j^L}}{\sum_k e^{cz_k^L}}, \tag{3.29}$$

where c is a positive constant. Note that $c = 1$ corresponds to the standard softmax function. But if we use a different value of c we get a different function, which is nonetheless qualitatively rather similar to the softmax. In particular, show that the output activations form a probability distribution, just as for the usual softmax. Suppose we allow c to become large, i.e., $c \to \infty$. What is the limiting value for the output activations a_j^L? After solving this problem it should be clear to you why we think of the $c = 1$ function as a "softened" version of the maximum function. This is the origin of the term "softmax".

- **Backpropagation with softmax and the log-likelihood cost** In the last chapter we derived the backpropagation algorithm for a network containing sigmoid layers. To apply the algorithm to a network with a softmax layer we need to figure out an expression for the error $\delta_j^L \equiv \partial C / \partial z_j^L$ in the final layer. Show that a suitable expression is:

$$\delta_j^L = a_j^L - y_j. \tag{3.30}$$

Using this expression we can apply the backpropagation algorithm to a network using a softmax output layer and the log-likelihood cost.

3.2 Overfitting and regularization

The Nobel prize winning physicist Enrico Fermi was once asked his opinion of a mathematical model some colleagues had proposed as the solution to an important unsolved physics problem. The model gave excellent agreement with experiment, but Fermi was skeptical. He asked how many free parameters could be set in the model. "Four" was the answer. Fermi replied[8] : "I remember my friend Johnny von Neumann used to say, with four parameters I can fit an elephant, and with five I can make him wiggle his trunk.".

The point, of course, is that models with a large number of free parameters can describe an amazingly wide range of phenomena. Even if such a model agrees well with the available data, that doesn't make it a good model. It may just mean there's enough freedom in the model that it can describe almost any data set of the given size, without capturing any genuine insights into the underlying phenomenon. When that happens the model will work well for the existing data, but will fail to generalize to new situations. The true test of a model is its ability to make predictions in situations it hasn't been exposed to before.

Fermi and von Neumann were suspicious of models with four parameters. Our 30 hidden neuron network for classifying MNIST digits has nearly 24,000 parameters! That's a lot of parameters. Our 100 hidden neuron network has nearly 80,000 parameters, and state-of-the-art deep neural nets sometimes contain millions or even billions of parameters. Should we trust the results?

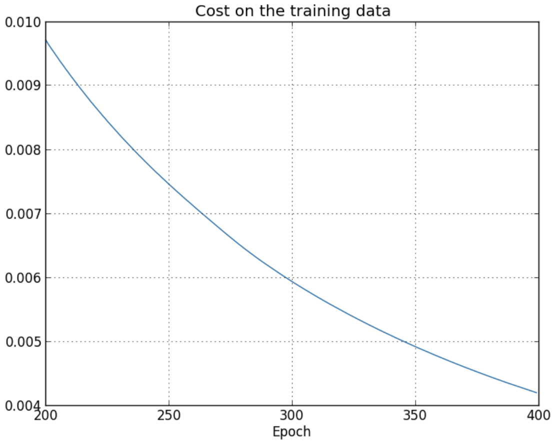
Let's sharpen this problem up by constructing a situation where our network does a bad job generalizing to new situations. We'll use our 30 hidden neuron network, with its 23,860 parameters. But we won't train the network using all 50,000 MNIST training images.

[8] The quote comes from a charming article by Freeman Dyson, who is one of the people who proposed the flawed model. A four-parameter elephant may be found here.

Instead, we'll use just the first 1,000 training images. Using that restricted set will make the problem with generalization much more evident. We'll train in a similar way to before, using the cross-entropy cost function, with a learning rate of $\eta = 0.5$ and a mini-batch size of 10. However, we'll train for 400 epochs, a somewhat larger number than before, because we're not using as many training examples. Let's use network2 to look at the way the cost function changes:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data[:1000], 400, 10, 0.5, evaluation_data=test_data,
      monitor_evaluation_accuracy=True, monitor_training_cost=True)
```

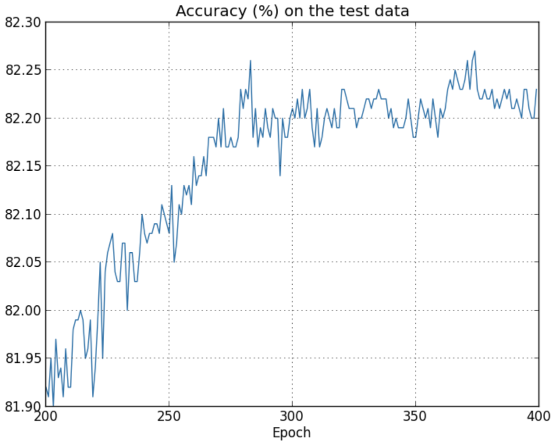
Using the results we can plot the way the cost changes as the network learns[9] :



This looks encouraging, showing a smooth decrease in the cost, just as we expect. Note that I've only shown training epochs 200 through 399. This gives us a nice up-close view of the later stages of learning, which, as we'll see, turns out to be where the interesting action is.

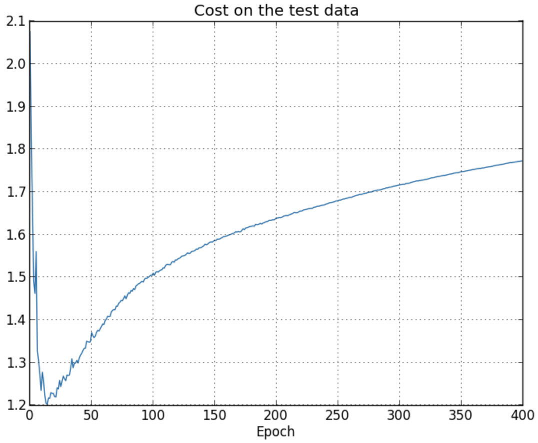
Let's now look at how the classification accuracy on the test data changes over time:

[9] This and the next four graphs were generated by the program overfitting .py.



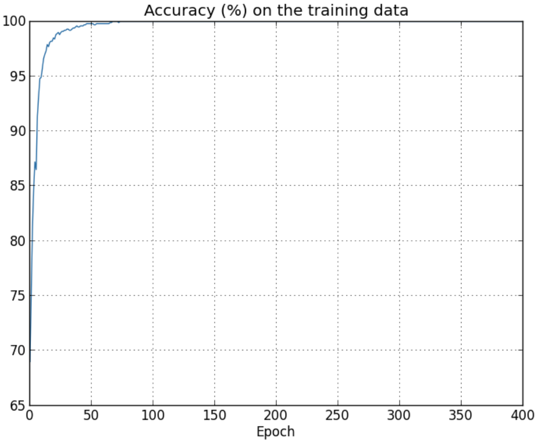
Again, I've zoomed in quite a bit. In the first 200 epochs (not shown) the accuracy rises to just under 82 percent. The learning then gradually slows down. Finally, at around epoch 280 the classification accuracy pretty much stops improving. Later epochs merely see small stochastic fluctuations near the value of the accuracy at epoch 280. Contrast this with the earlier graph, where the cost associated to the training data continues to smoothly drop. If we just look at that cost, it appears that our model is still getting "better". But the test accuracy results show the improvement is an illusion. Just like the model that Fermi disliked, what our network learns after epoch 280 no longer generalizes to the test data. And so it's not useful learning. We say the network is *overfitting* or *overtraining* beyond epoch 280.

You might wonder if the problem here is that I'm looking at the cost on the training data, as opposed to the *classification accuracy* on the test data. In other words, maybe the problem is that we're making an apples and oranges comparison. What would happen if we compared the cost on the training data with the cost on the test data, so we're comparing similar measures? Or perhaps we could compare the classification accuracy on both the training data and the test data? In fact, essentially the same phenomenon shows up no matter how we do the comparison. The details do change, however. For instance, let's look at the cost on the test data:



We can see that the cost on the test data improves until around epoch 15, but after that it actually starts to get worse, even though the cost on the training data is continuing to get better. This is another sign that our model is overfitting. It poses a puzzle, though, which is whether we should regard epoch 15 or epoch 280 as the point at which overfitting is coming to dominate learning? From a practical point of view, what we really care about is improving classification accuracy on the test data, while the cost on the test data is no more than a proxy for classification accuracy. And so it makes most sense to regard epoch 280 as the point beyond which overfitting is dominating learning in our neural network.

Another sign of overfitting may be seen in the classification accuracy on the training data:



The accuracy rises all the way up to 100 percent. That is, our network correctly classifies all 1,000 training images! Meanwhile, our test accuracy tops out at just 82.27 percent. So our network really is learning about peculiarities of the training set, not just recognizing digits in general. It's almost as though our network is merely memorizing the training set, without understanding digits well enough to generalize to the test set.

Overfitting is a major problem in neural networks. This is especially true in modern networks, which often have very large numbers of weights and biases. To train effectively, we need a way of detecting when overfitting is going on, so we don't overtrain. And we'd like to have techniques for reducing the effects of overfitting.

The obvious way to detect overfitting is to use the approach above, keeping track of accuracy on the test data as our network trains. If we see that the accuracy on the test data is no longer improving, then we should stop training. Of course, strictly speaking, this is not necessarily a sign of overfitting. It might be that accuracy on the test data and the training data both stop improving at the same time. Still, adopting this strategy will prevent overfitting.

In fact, we'll use a variation on this strategy. Recall that when we load in the MNIST data we load in three data sets:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
```

Up to now we've been using the training_data and test_data, and ignoring the validation_data . The validation_data contains 10,000 images of digits, images which are different from the 50,000 images in the MNIST training set, and the 10,000 images in the MNIST test set. Instead of using the test_data to prevent overfitting, we will use the validation_data. To do this, we'll use much the same strategy as was described above for the test_data. That is, we'll compute the classification accuracy on the validation_data at the end of each epoch. Once the classification accuracy on the validation_data has saturated, we stop training. This strategy is called early stopping. Of course, in practice we won't immediately know when the accuracy has saturated. Instead, we continue training until we're confident that the accuracy has saturated[10].

Why use the validation_data to prevent overfitting, rather than the test_data? In fact, this is part of a more general strategy, which is to use the validation_data to evaluate different trial choices of hyper-parameters such as the number of epochs to train for, the learning rate, the best network architecture, and so on. We use such evaluations to find and set good values for the hyper-parameters. Indeed, although I haven't mentioned it until now, that is, in part, how I arrived at the hyper-parameter choices made earlier in this book. (More on this later.)

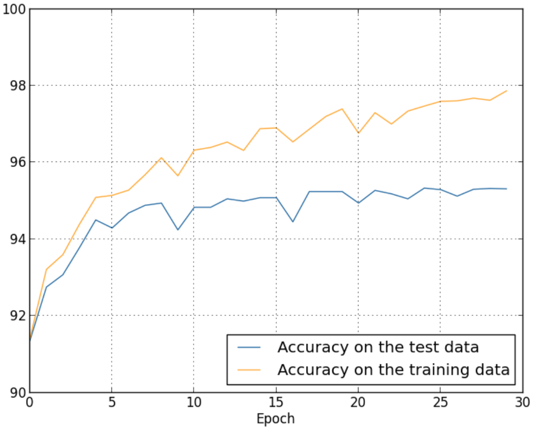
Of course, that doesn't in any way answer the question of why we're using the validation_data to prevent overfitting, rather than the test_data. Instead, it replaces it with a more general question, which is why we're using the validation_data rather than the test_data to set good hyper-parameters? To understand why, consider that when setting hyper-parameters we're likely to try many different choices for the hyper-parameters. If we set the hyper-parameters based on evaluations of the test_data it's possible we'll end up overfitting our hyper-parameters to the test_data. That is, we may end up finding hyper-parameters which fit particular peculiarities of the test_data, but where the performance of the network won't generalize to other data sets. We guard against that by figuring out the hyper-parameters using the validation_data. Then, once we've got the hyper-parameters we want, we do a final evaluation of accuracy using the test_data. That gives us confidence that our results on the test_data are a true measure of how well our neural network generalizes. To put it another way, you can think of the validation data as a type of training data that helps us learn good hyper-parameters. This approach to finding good hyper-parameters is sometimes

[10] It requires some judgment to determine when to stop. In my earlier graphs I identified epoch 280 as the place at which accuracy saturated. It's possible that was too pessimistic. Neural networks sometimes plateau for a while in training, before continuing to improve. I wouldn't be surprised if more learning could have occurred even after epoch 400, although the magnitude of any further improvement would likely be small. So it's possible to adopt more or less aggressive strategies for early stopping.

known as the hold out method, since the validation_data is kept apart or "held out" from the training_data.

Now, in practice, even after evaluating performance on the test_data we may change our minds and want to try another approach – perhaps a different network architecture – which will involve finding a new set of hyper-parameters. If we do this, isn't there a danger we'll end up overfitting to the test_data as well? Do we need a potentially infinite regress of data sets, so we can be confident our results will generalize? Addressing this concern fully is a deep and difficult problem. But for our practical purposes, we're not going to worry too much about this question. Instead, we'll plunge ahead, using the basic hold out method, based on the training_data, validation_data, and test_data, as described above.

We've been looking so far at overfitting when we're just using 1,000 training images. What happens when we use the full training set of 50,000 images? We'll keep all the other parameters the same (30 hidden neurons, learning rate 0.5, mini-batch size of 10), but train using all 50,000 images for 30 epochs. Here's a graph showing the results for the classification accuracy on both the training data and the test data. Note that I've used the test data here, rather than the validation data, in order to make the results more directly comparable with the earlier graphs.



As you can see, the accuracy on the test and training data remain much closer together than when we were using 1,000 training examples. In particular, the best classification accuracy of 97.86 percent on the training data is only 2.53 percent higher than the 95.33 percent on the test data. That's compared to the 17.73 percent gap we had earlier! Overfitting is still going on, but it's been greatly reduced. Our network is generalizing much better from the training data to the test data. In general, one of the best ways of reducing overfitting is to increase the size of the training data. With enough training data it is difficult for even a very large network to overfit. Unfortunately, training data can be expensive or difficult to acquire, so this is not always a practical option.

3.2.1 Regularization

Increasing the amount of training data is one way of reducing overfitting. Are there other ways we can reduce the extent to which overfitting occurs? One possible approach is to reduce the size of our network. However, large networks have the potential to be more powerful than small networks, and so this is an option we'd only adopt reluctantly.

Fortunately, there are other techniques which can reduce overfitting, even when we have a fixed network and fixed training data. These are known as *regularization* techniques. In this section I describe one of the most commonly used regularization techniques, a technique sometimes known as *weight decay* or *L2 regularization*. The idea of L2 regularization is to add an extra term to the cost function, a term called the *regularization term*. Here's the regularized cross-entropy:

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right] + \frac{\lambda}{2n}\sum_w w^2. \tag{3.31}$$

The first term is just the usual expression for the cross-entropy. But we've added a second term, namely the sum of the squares of all the weights in the network. This is scaled by a factor $\lambda/2n$, where $\lambda > 0$ is known as the *regularization parameter*, and n is, as usual, the size of our training set. I'll discuss later how λ is chosen. It's also worth noting that the regularization term doesn't include the biases. I'll also come back to that below.

Of course, it's possible to regularize other cost functions, such as the quadratic cost. This can be done in a similar way:

$$C = \frac{1}{2n}\sum_x \|y - a^L\|^2 + \frac{\lambda}{2n}\sum_w w^2. \tag{3.32}$$

In both cases we can write the regularized cost function as

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2, \tag{3.33}$$

where C_0 is the original, unregularized cost function.

Intuitively, the effect of regularization is to make it so the network prefers to learn small weights, all other things being equal. Large weights will only be allowed if they considerably improve the first part of the cost function. Put another way, regularization can be viewed as a way of compromising between finding small weights and minimizing the original cost function. The relative importance of the two elements of the compromise depends on the value of λ: when λ is small we prefer to minimize the original cost function, but when λ is large we prefer small weights.

Now, it's really not at all obvious why making this kind of compromise should help reduce overfitting! But it turns out that it does. We'll address the question of why it helps in the next section. But first, let's work through an example showing that regularization really does reduce overfitting.

To construct such an example, we first need to figure out how to apply our stochastic gradient descent learning algorithm in a regularized neural network. In particular, we need to know how to compute the partial derivatives $\partial C/\partial w$ and $\partial C/\partial b$ for all the weights and biases in the network. Taking the partial derivatives of Equation (3.33) gives

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n}w \tag{3.34}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}. \tag{3.35}$$

The $\partial C_0/\partial w$ and $\partial C_0/\partial b$ terms can be computed using backpropagation, as described in the last chapter. And so we see that it's easy to compute the gradient of the regularized cost function: just use backpropagation, as usual, and then add $\frac{\lambda}{n}w$ to the partial derivative of all the weight terms. The partial derivatives with respect to the biases are unchanged, and so the gradient descent learning rule for the biases doesn't change from the usual rule:

$$b \rightarrow b - \eta \frac{\partial C_0}{\partial b}. \tag{3.36}$$

The learning rule for the weights becomes:

$$w \quad \rightarrow \quad w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{n} w = \left(1 - \frac{\eta\lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}. \tag{3.37}$$

This is exactly the same as the usual gradient descent learning rule, except we first rescale the weight w by a factor $1\eta\frac{\lambda}{n}$. This rescaling is sometimes referred to as *weight decay*, since it makes the weights smaller. At first glance it looks as though this means the weights are being driven unstoppably toward zero. But that's not right, since the other term may lead the weights to increase, if so doing causes a decrease in the unregularized cost function.

Okay, that's how gradient descent works. What about stochastic gradient descent? Well, just as in unregularized stochastic gradient descent, we can estimate $\partial C_0/\partial w$ by averaging over a mini-batch of m training examples. Thus the regularized learning rule for stochastic gradient descent becomes (c.f. Equation (1.20))

$$w \rightarrow \left(1 - \frac{\eta\lambda}{n}\right) w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w}, \tag{3.38}$$

where the sum is over training examples x in the mini-batch, and C_x is the (unregularized) cost for each training example. This is exactly the same as the usual rule for stochastic gradient descent, except for the $1\eta\lambda/n$ weight decay factor. Finally, and for completeness, let me state the regularized learning rule for the biases. This is, of course, exactly the same as in the unregularized case (c.f. Equation (1.21)),

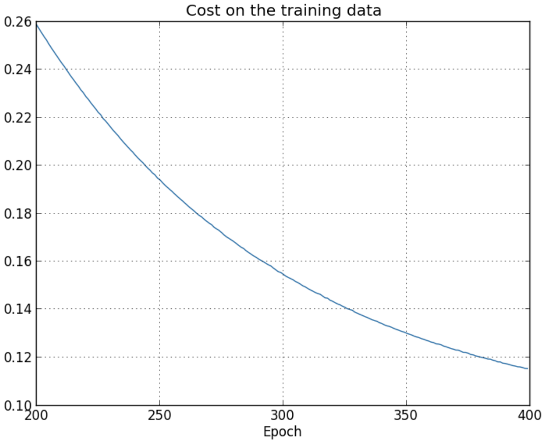
$$b \rightarrow b - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial b}, \tag{3.39}$$

where the sum is over training examples x in the mini-batch.

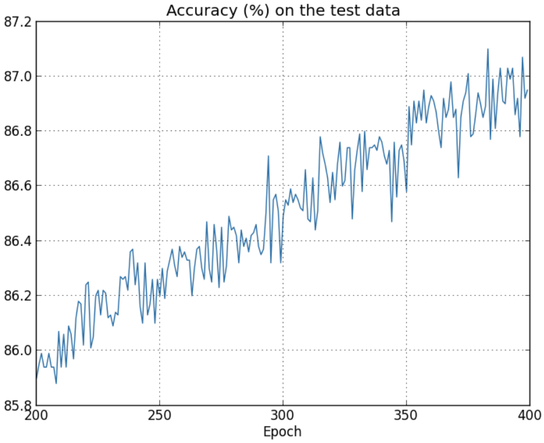
Let's see how regularization changes the performance of our neural network. We'll use a network with 30 hidden neurons, a mini-batch size of 10, a learning rate of 0.5, and the cross-entropy cost function. However, this time we'll use a regularization parameter of $\lambda = 0.1$. Note that in the code, we use the variable name `lmbda`, because `lambda` is a reserved word in Python, with an unrelated meaning. I've also used the `test_data` again, not the `validation_data`. Strictly speaking, we should use the `validation_data`, for all the reasons we discussed earlier. But I decided to use the `test_data` because it makes the results more directly comparable with our earlier, unregularized results. You can easily change the code to use the `validation_data` instead, and you'll find that it gives similar results.

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data[:1000], 400, 10, 0.5, evaluation_data=test_data, lmbda = 0.1, \
... monitor_evaluation_cost=True, monitor_evaluation_accuracy=True, monitor_training_cost
    =True, monitor_training_accuracy=True)
```

The cost on the training data decreases over the whole time, much as it did in the earlier, unregularized case[11]:



But this time the accuracy on the `test_data` continues to increase for the entire 400 epochs:



Clearly, the use of regularization has suppressed overfitting. What's more, the accuracy is considerably higher, with a peak classification accuracy of 87.1 percent, compared to the peak of 82.27 percent obtained in the unregularized case. Indeed, we could almost certainly get considerably better results by continuing to train past 400 epochs. It seems that, empirically, regularization is causing our network to generalize better, and considerably reducing the effects of overfitting.

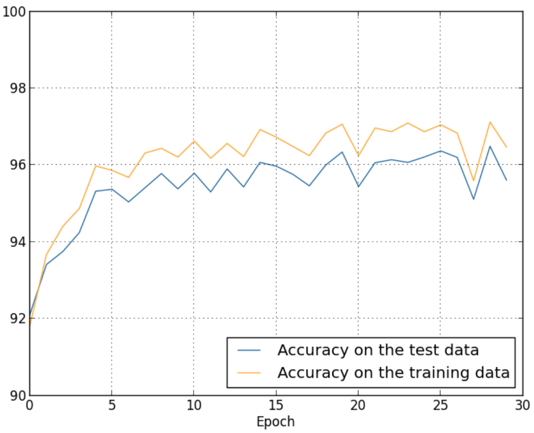
[11] This and the next two graphs were produced with the program `overfitting.py`.

What happens if we move out of the artificial environment of just having 1,000 training images, and return to the full 50,000 image training set? Of course, we've seen already that overfitting is much less of a problem with the full 50,000 images. Does regularization help any further? Let's keep the hyper-parameters the same as before – 30 epochs, learning rate 0.5, mini-batch size of 10. However, we need to modify the regularization parameter. The reason is because the size n of the training set has changed from n=1,000 to n=50,000, and this changes the weight decay factor $1\eta\lambda/n$. If we continued to use $\lambda = 0.1$ that would mean much less weight decay, and thus much less of a regularization effect. We compensate by changing to $\lambda = 5.0$.

Okay, let's train our network, stopping first to re-initialize the weights:

```
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data, lmbda = 5.0,
... monitor_evaluation_accuracy=True, monitor_training_accuracy=True)
```

We obtain the results:



There's lots of good news here. First, our classification accuracy on the test data is up, from 95.49 percent when running unregularized, to 96.49 percent. That's a big improvement. Second, we can see that the gap between results on the training and test data is much narrower than before, running at under a percent. That's still a significant gap, but we've obviously made substantial progress reducing overfitting.

Finally, let's see what test classification accuracy we get when we use 100 hidden neurons and a regularization parameter of $\lambda = 5.0$. I won't go through a detailed analysis of overfitting here, this is purely for fun, just to see how high an accuracy we can get when we use our new tricks: the cross-entropy cost function and L2 regularization.

```
>>> net = network2.Network([784, 100, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5, lmbda=5.0, evaluation_data=validation_data,
... monitor_evaluation_accuracy=True)
```

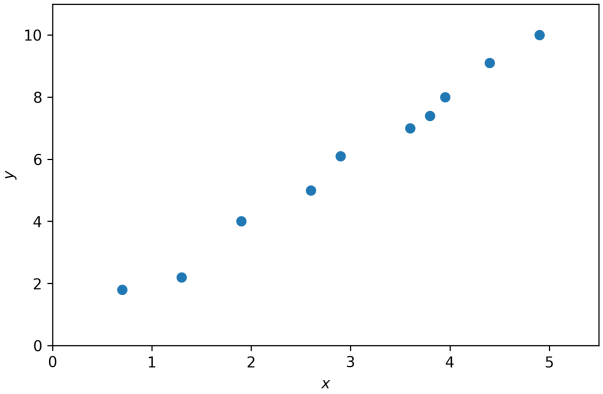
The final result is a classification accuracy of 97.92 percent on the validation data. That's a big jump from the 30 hidden neuron case. In fact, tuning just a little more, to run for 60 epochs at $\eta = 0.1$ and $\lambda = 5.0$ we break the 98 percent barrier, achieving 98.04 percent classification accuracy on the validation data. Not bad for what turns out to be 152 lines of code!

I've described regularization as a way to reduce overfitting and to increase classification accuracies. In fact, that's not the only benefit. Empirically, when doing multiple runs of our MNIST networks, but with different (random) weight initializations, I've found that the unregularized runs will occasionally get "stuck", apparently caught in local minima of the cost function. The result is that different runs sometimes provide quite different results. By contrast, the regularized runs have provided much more easily replicable results.

Why is this going on? Heuristically, if the cost function is unregularized, then the length of the weight vector is likely to grow, all other things being equal. Over time this can lead to the weight vector being very large indeed. This can cause the weight vector to get stuck pointing in more or less the same direction, since changes due to gradient descent only make tiny changes to the direction, when the length is long. I believe this phenomenon is making it hard for our learning algorithm to properly explore the weight space, and consequently harder to find good minima of the cost function.

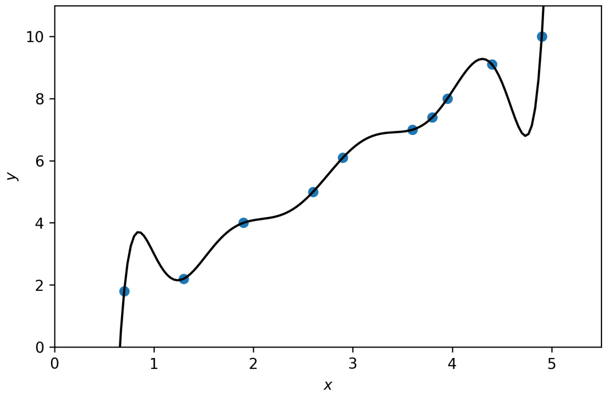
3.2.2 Why does regularization help reduce overfitting?

We've seen empirically that regularization helps reduce overfitting. That's encouraging but, unfortunately, it's not obvious why regularization helps! A standard story people tell to explain what's going on is along the following lines: smaller weights are, in some sense, lower complexity, and so provide a simpler and more powerful explanation for the data, and should thus be preferred. That's a pretty terse story, though, and contains several elements that perhaps seem dubious or mystifying. Let's unpack the story and examine it critically. To do that, let's suppose we have a simple data set for which we wish to build a model:

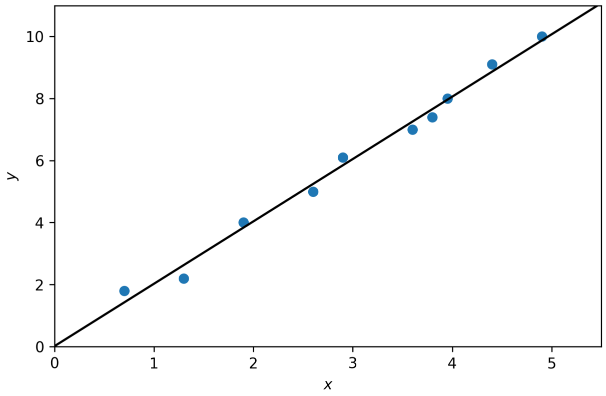


Implicitly, we're studying some real-world phenomenon here, with x and y representing real-world data. Our goal is to build a model which lets us predict y as a function of x. We could try using neural networks to build such a model, but I'm going to do something even simpler: I'll try to model y as a polynomial in x. I'm doing this instead of using neural nets because using polynomials will make things particularly transparent. Once we've understood the polynomial case, we'll translate to neural networks. Now, there are ten points in the graph above, which means we can find a unique 9-th-order polynomial $y = a_0x^9 + a_1x^8 + \ldots + a_9$ which fits the data exactly. Here's the graph of that polynomial[12] :

[12] I won't show the coefficients explicitly, although they are easy to find using a routine such as Numpy's `polyfit`.



That provides an exact fit. But we can also get a good fit using the linear model $y = 2x$:



Which of these is the better model? Which is more likely to be true? And which model is more likely to generalize well to other examples of the same underlying real-world phenomenon?

These are difficult questions. In fact, we can't determine with certainty the answer to any of the above questions, without much more information about the underlying real-world phenomenon. But let's consider two possibilities: (1) the 9th order polynomial is, in fact, the model which truly describes the real-world phenomenon, and the model will therefore generalize perfectly; (2) the correct model is $y = 2x$, but there's a little additional noise due to, say, measurement error, and that's why the model isn't an exact fit.

It's not *a priori* possible to say which of these two possibilities is correct. (Or, indeed, if some third possibility holds). Logically, either could be true. And it's not a trivial difference. It's true that on the data provided there's only a small difference between the two models. But suppose we want to predict the value of y corresponding to some large value of x, much larger than any shown on the graph above. If we try to do that there will be a dramatic difference between the predictions of the two models, as the 9th order polynomial model comes to be dominated by the x^9 term, while the linear model remains, well, linear.

One point of view is to say that in science we should go with the simpler explanation, unless compelled not to. When we find a simple model that seems to explain many data points we are tempted to shout "Eureka!" After all, it seems unlikely that a simple explanation should occur merely by coincidence. Rather, we suspect that the model must be expressing some underlying truth about the phenomenon. In the case at hand, the model $y = 2x + \text{noise}$ seems much simpler than $y = a_0x^9 + a_1x^8 + \ldots$. It would be surprising if that simplicity had occurred by chance, and so we suspect that $y = 2x + \text{noise}$ expresses some underlying truth. In this point of view, the 9th order model is really just learning the effects of local noise. And so while the 9th order model works perfectly for these particular data points, the model will fail to generalize to other data points, and the noisy linear model will have greater predictive power.

Let's see what this point of view means for neural networks. Suppose our network mostly has small weights, as will tend to happen in a regularized network. The smallness of the weights means that the behaviour of the network won't change too much if we change a few random inputs here and there. That makes it difficult for a regularized network to learn the effects of local noise in the data. Think of it as a way of making it so single pieces of evidence don't matter too much to the output of the network. Instead, a regularized network learns to respond to types of evidence which are seen often across the training set. By contrast, a network with large weights may change its behaviour quite a bit in response to small changes in the input. And so an unregularized network can use large weights to learn a complex model that carries a lot of information about the noise in the training data. In a nutshell, regularized networks are constrained to build relatively simple models based on patterns seen often in the training data, and are resistant to learning peculiarities of the noise in the training data. The hope is that this will force our networks to do real learning about the phenomenon at hand, and to generalize better from what they learn.

With that said, this idea of preferring simpler explanation should make you nervous. People sometimes refer to this idea as "Occam's Razor", and will zealously apply it as though it has the status of some general scientific principle. But, of course, it's not a general scientific principle. There is no *a priori* logical reason to prefer simple explanations over more complex explanations. Indeed, sometimes the more complex explanation turns out to be correct.

Let me describe two examples where more complex explanations have turned out to be correct. In the 1940s the physicist Marcel Schein announced the discovery of a new particle of nature. The company he worked for, General Electric, was ecstatic, and publicized the discovery widely. But the physicist Hans Bethe was skeptical. Bethe visited Schein, and looked at the plates showing the tracks of Schein's new particle. Schein showed Bethe plate after plate, but on each plate Bethe identified some problem that suggested the data should be discarded. Finally, Schein showed Bethe a plate that looked good. Bethe said it might just be a statistical fluke. Schein: "Yes, but the chance that this would be statistics, even according to your own formula, is one in five." Bethe: "But we have already looked at five plates." Finally, Schein said: "But on my plates, each one of the good plates, each one of the good pictures, you explain by a different theory, whereas I have one hypothesis that explains all the plates, that they are [the new particle]." Bethe replied: "The sole difference between your and my explanations is that yours is wrong and all of mine are right. Your single explanation is wrong, and all of my multiple explanations are right." Subsequent work confirmed that Nature agreed with Bethe, and Schein's particle is no more[13].

As a second example, in 1859 the astronomer Urbain Le Verrier observed that the orbit of the planet Mercury doesn't have quite the shape that Newton's theory of gravitation says it should have. It was a tiny, tiny deviation from Newton's theory, and several of the explanations proferred at the time boiled down to saying that Newton's theory was more or less right, but needed a tiny alteration. In 1916, Einstein showed that the deviation could be explained very well using his general theory of relativity, a theory radically different to Newtonian gravitation,

[13] The story is related by the physicist Richard Feynman in an [link]interview with the historian Charles Weiner.

and based on much more complex mathematics. Despite that additional complexity, today it's accepted that Einstein's explanation is correct, and Newtonian gravity, even in its modified forms, is wrong. This is in part because we now know that Einstein's theory explains many other phenomena which Newton's theory has difficulty with. Furthermore, and even more impressively, Einstein's theory accurately predicts several phenomena which aren't predicted by Newtonian gravity at all. But these impressive qualities weren't entirely obvious in the early days. If one had judged merely on the grounds of simplicity, then some modified form of Newton's theory would arguably have been more attractive.

There are three morals to draw from these stories. First, it can be quite a subtle business deciding which of two explanations is truly "simpler". Second, even if we can make such a judgment, simplicity is a guide that must be used with great caution! Third, the true test of a model is not simplicity, but rather how well it does in predicting new phenomena, in new regimes of behaviour.

With that said, and keeping the need for caution in mind, it's an empirical fact that regularized neural networks usually generalize better than unregularized networks. And so through the remainder of the book we will make frequent use of regularization. I've included the stories above merely to help convey why no-one has yet developed an entirely convincing theoretical explanation for why regularization helps networks generalize. Indeed, researchers continue to write papers where they try different approaches to regularization, compare them to see which works better, and attempt to understand why different approaches work better or worse. And so you can view regularization as something of a kludge. While it often helps, we don't have an entirely satisfactory systematic understanding of what's going on, merely incomplete heuristics and rules of thumb.

There's a deeper set of issues here, issues which go to the heart of science. It's the question of how we generalize. Regularization may give us a computational magic wand that helps our networks generalize better, but it doesn't give us a principled understanding of how generalization works, nor of what the best approach is[14].

This is particularly galling because in everyday life, we humans generalize phenomenally well. Shown just a few images of an elephant a child will quickly learn to recognize other elephants. Of course, they may occasionally make mistakes, perhaps confusing a rhinoceros for an elephant, but in general this process works remarkably accurately. So we have a system - the human brain - with a huge number of free parameters. And after being shown just one or a few training images that system learns to generalize to other images. Our brains are, in some sense, regularizing amazingly well! How do we do it? At this point we don't know. I expect that in years to come we will develop more powerful techniques for regularization in artificial neural networks, techniques that will ultimately enable neural nets to generalize well even from small data sets.

In fact, our networks already generalize better than one might a priori expect. A network with 100 hidden neurons has nearly 80,000 parameters. We have only 50,000 images in our training data. It's like trying to fit an 80,000th degree polynomial to 50,000 data points. By all rights, our network should overfit terribly. And yet, as we saw earlier, such a network actually does a pretty good job generalizing. Why is that the case? It's not well understood. It has been conjectured[15] that "the dynamics of gradient descent learning in multilayer nets has a 'self-regularization' effect". This is exceptionally fortunate, but it's also somewhat disquieting that we don't understand why it's the case. In the meantime, we will adopt the

[14] These issues go back to the problem of induction, famously discussed by the Scottish philosopher David Hume in "An Enquiry Concerning Human Understanding" (1748). The problem of induction has been given a modern machine learning form in the no-free lunch theorem (link) of David Wolpert and William Macready (1997).

[15] In Gradient-Based Learning Applied to Document Recognition, by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998).

pragmatic approach and use regularization whenever we can. Our neural networks will be the better for it.

Let me conclude this section by returning to a detail which I left unexplained earlier: the fact that L2 regularization doesn't constrain the biases. Of course, it would be easy to modify the regularization procedure to regularize the biases. Empirically, doing this often doesn't change the results very much, so to some extent it's merely a convention whether to regularize the biases or not. However, it's worth noting that having a large bias doesn't make a neuron sensitive to its inputs in the same way as having large weights. And so we don't need to worry about large biases enabling our network to learn the noise in our training data. At the same time, allowing large biases gives our networks more flexibility in behaviour - in particular, large biases make it easier for neurons to saturate, which is sometimes desirable. For these reasons we don't usually include bias terms when regularizing.

3.2.3 Other techniques for regularization

There are many regularization techniques other than L2 regularization. In fact, so many techniques have been developed that I can't possibly summarize them all. In this section I briefly describe three other approaches to reducing overfitting: L1 regularization, dropout, and artificially increasing the training set size. We won't go into nearly as much depth studying these techniques as we did earlier. Instead, the purpose is to get familiar with the main ideas, and to appreciate something of the diversity of regularization techniques available.

L1 regularization: In this approach we modify the unregularized cost function by adding the sum of the absolute values of the weights:

$$C = C_0 + \frac{\lambda}{n}\sum_w |w|. \tag{3.40}$$

Intuitively, this is similar to L2 regularization, penalizing large weights, and tending to make the network prefer small weights. Of course, the L1 regularization term isn't the same as the L2 regularization term, and so we shouldn't expect to get exactly the same behaviour. Let's try to understand how the behaviour of a network trained using L1 regularization differs from a network trained using L2 regularization.

To do that, we'll look at the partial derivatives of the cost function. Differentiating (95) we obtain:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n}\,\mathrm{sgn}(w), \tag{3.41}$$

where $\mathrm{sgn}(w)$ is the sign of w, that is, $+1$ if w is positive, and -1 if w is negative. Using this expression, we can easily modify backpropagation to do stochastic gradient descent using L1 regularization. The resulting update rule for an L1 regularized network is

$$w \rightarrow w' = w - \frac{\eta\lambda}{n}\mathrm{sgn}(w) - \eta\frac{\partial C_0}{\partial w}, \tag{3.42}$$

where, as per usual, we can estimate $\partial C_0/\partial w$ using a mini-batch average, if we wish.

Compare that to the update rule for L2 regularization (c.f. Equation (3.38)),

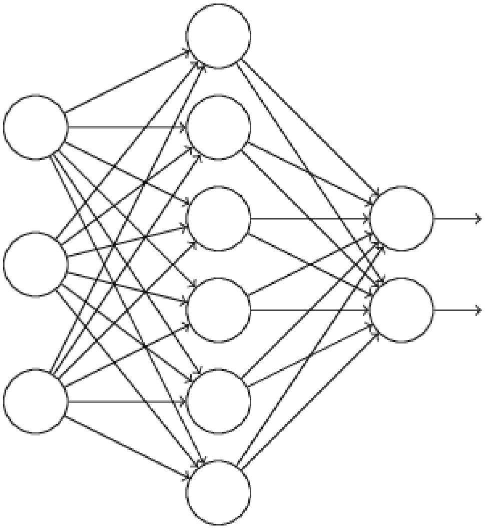
$$w \rightarrow w' = w\left(1 - \frac{\eta\lambda}{n}\right) - \eta\frac{\partial C_0}{\partial w}. \tag{3.43}$$

In both expressions the effect of regularization is to shrink the weights. This accords with our intuition that both kinds of regularization penalize large weights. But the way the weights shrink is different. In L1 regularization, the weights shrink by a constant amount toward 0. In L2 regularization, the weights shrink by an amount which is proportional to w. And so when a particular weight has a large magnitude, $|w|$, L1 regularization shrinks the weight much less than L2 regularization does. By contrast, when $|w|$ is small, L1 regularization shrinks the weight much more than L2 regularization. The net result is that L1 regularization tends to concentrate the weight of the network in a relatively small number of high-importance connections, while the other weights are driven toward zero.

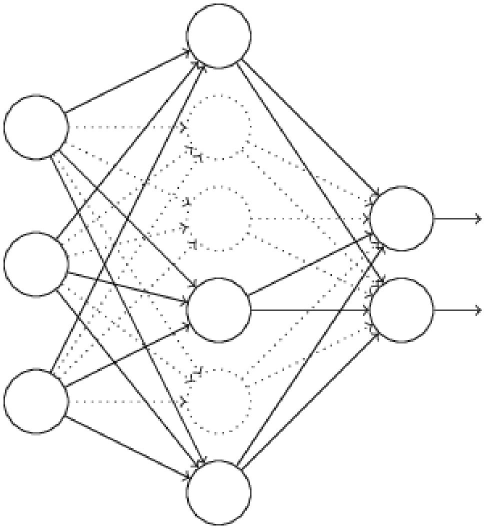
I've glossed over an issue in the above discussion, which is that the partial derivative $\partial C/\partial w$ isn't defined when w=0. The reason is that the function $|w|$ has a sharp "corner" at $w = 0$, and so isn't differentiable at that point. That's okay, though. What we'll do is just apply the usual (unregularized) rule for stochastic gradient descent when $w = 0$. That should be okay – intuitively, the effect of regularization is to shrink weights, and obviously it can't shrink a weight which is already 0. To put it more precisely, we'll use Equations (3.41) and (3.42) with the convention that $rmsgn(0) = 0$. That gives a nice, compact rule for doing stochastic gradient descent with L1 regularization.

Dropout: Dropout is a radically different technique for regularization. Unlike L1 and L2 regularization, dropout doesn't rely on modifying the cost function. Instead, in dropout we modify the network itself. Let me describe the basic mechanics of how dropout works, before getting into why it works, and what the results are.

Suppose we're trying to train a network:



In particular, suppose we have a training input x and corresponding desired output y. Ordinarily, we'd train by forward-propagating x through the network, and then backpropagating to determine the contribution to the gradient. With dropout, this process is modified. We start by randomly (and temporarily) deleting half the hidden neurons in the network, while leaving the input and output neurons untouched. After doing this, we'll end up with a network along the following lines. Note that the dropout neurons, i.e., the neurons which have been temporarily deleted, are still ghosted in:



We forward-propagate the input x through the modified network, and then backpropagate the result, also through the modified network. After doing this over a mini-batch of examples, we update the appropriate weights and biases. We then repeat the process, first restoring the dropout neurons, then choosing a new random subset of hidden neurons to delete, estimating the gradient for a different mini-batch, and updating the weights and biases in the network.

By repeating this process over and over, our network will learn a set of weights and biases. Of course, those weights and biases will have been learnt under conditions in which half the hidden neurons were dropped out. When we actually run the full network that means that twice as many hidden neurons will be active. To compensate for that, we halve the weights outgoing from the hidden neurons.

This dropout procedure may seem strange and *ad hoc*. Why would we expect it to help with regularization? To explain what's going on, I'd like you to briefly stop thinking about dropout, and instead imagine training neural networks in the standard way (no dropout). In particular, imagine we train several different neural networks, all using the same training data. Of course, the networks may not start out identical, and as a result after training they may sometimes give different results. When that happens we could use some kind of averaging or voting scheme to decide which output to accept. For instance, if we have trained five networks, and three of them are classifying a digit as a “3”, then it probably really is a “3”. The other two networks are probably just making a mistake. This kind of averaging scheme is often found to be a powerful (though expensive) way of reducing overfitting. The reason is that the different networks may overfit in different ways, and averaging may help eliminate that kind of overfitting.

What's this got to do with dropout? Heuristically, when we dropout different sets of neurons, it's rather like we're training different neural networks. And so the dropout procedure is like averaging the effects of a very large number of different networks. The different networks will overfit in different ways, and so, hopefully, the net effect of dropout will be to reduce overfitting.

A related heuristic explanation for dropout is given in one of the earliest papers to use the technique[16]: “This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.” In other words, if we think of our network as a model which is making predictions, then we can think of dropout as a way of making sure that the model is robust to the loss of any individual piece of evidence. In this, it’s somewhat similar to L1 and L2 regularization, which tend to reduce weights, and thus make the network more robust to losing any individual connection in the network.

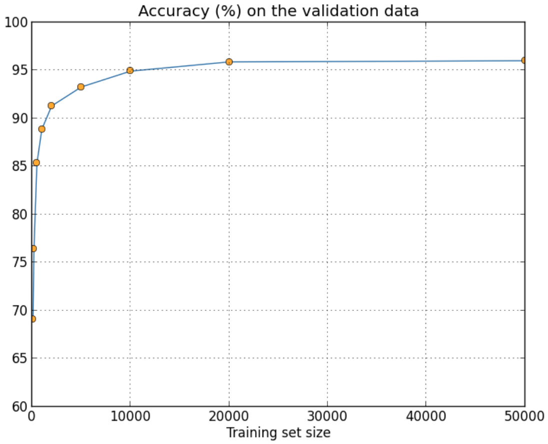
Of course, the true measure of dropout is that it has been very successful in improving the performance of neural networks. The original paper[17] introducing the technique applied it to many different tasks. For us, it’s of particular interest that they applied dropout to MNIST digit classification, using a vanilla feedforward neural network along lines similar to those we’ve been considering. The paper noted that the best result anyone had achieved up to that point using such an architecture was 98.4 percent classification accuracy on the test set. They improved that to 98.7 percent accuracy using a combination of dropout and a modified form of L2 regularization. Similarly impressive results have been obtained for many other tasks, including problems in image and speech recognition, and natural language processing. Dropout has been especially useful in training large, deep networks, where the problem of overfitting is often acute.

Artificially expanding the training data: We saw earlier that our MNIST classification accuracy dropped down to percentages in the mid-80s when we used only 1,000 training images. It’s not surprising that this is the case, since less training data means our network will be exposed to fewer variations in the way human beings write digits. Let’s try training our 30 hidden neuron network with a variety of different training data set sizes, to see how performance varies. We train using a mini-batch size of 10, a learning rate $\eta = 0.5$, a regularization parameter $\lambda = 5.0$, and the cross-entropy cost function. We will train for 30 epochs when the full training data set is used, and scale up the number of epochs proportionally when smaller training sets are used. To ensure the weight decay factor remains the same across training sets, we will use a regularization parameter of $\lambda = 5.0$ when the full training data set is used, and scale down λ proportionally when smaller training sets are used[18].

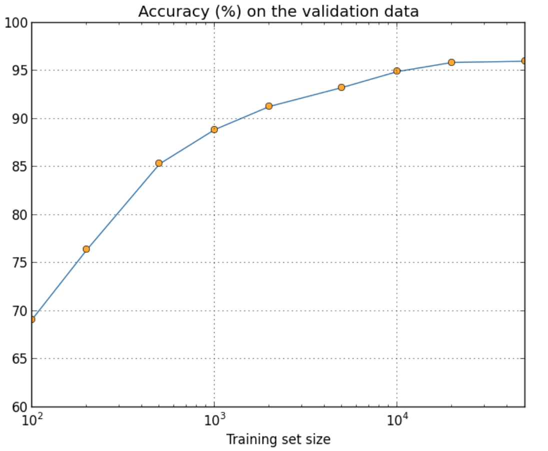
[16] ImageNet Classification with Deep Convolutional Neural Networks, by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton (2012).

[17] Improving neural networks by preventing co-adaptation of feature detectors by Geoffrey Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2012). Note that the paper discusses a number of subtleties that I have glossed over in this brief introduction.

[18] This and the next two graph are produced with the program `more_data.py`.

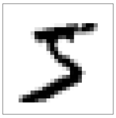


As you can see, the classification accuracies improve considerably as we use more training data. Presumably this improvement would continue still further if more data was available. Of course, looking at the graph above it does appear that we're getting near saturation. Suppose, however, that we redo the graph with the training set size plotted logarithmically:

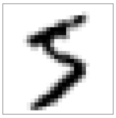


It seems clear that the graph is still going up toward the end. This suggests that if we used vastly more training data – say, millions or even billions of handwriting samples, instead of just 50,000 – then we'd likely get considerably better performance, even from this very small network.

Obtaining more training data is a great idea. Unfortunately, it can be expensive, and so is not always possible in practice. However, there's another idea which can work nearly as well, and that's to artificially expand the training data. Suppose, for example, that we take an MNIST training image of a five,



and rotate it by a small amount, let's say 15 degrees:



It's still recognizably the same digit. And yet at the pixel level it's quite different to any image currently in the MNIST training data. It's conceivable that adding this image to the training data might help our network learn more about how to classify digits. What's more, obviously we're not limited to adding just this one image. We can expand our training data by making *many* small rotations of *all* the MNIST training images, and then using the expanded training data to improve our network's performance.

This idea is very powerful and has been widely used. Let's look at some of the results from a paper[19] which applied several variations of the idea to MNIST. One of the neural network architectures they considered was along similar lines to what we've been using, a feedforward network with 800 hidden neurons and using the cross-entropy cost function. Running the network with the standard MNIST training data they achieved a classification accuracy of 98.4 percent on their test set. But then they expanded the training data, using not just rotations, as I described above, but also translating and skewing the images. By training on the expanded data set they increased their network's accuracy to 98.9 percent. They also experimented with what they called "elastic distortions", a special type of image distortion intended to emulate the random oscillations found in hand muscles. By using the elastic distortions to expand the data they achieved an even higher accuracy, 99.3 percent. Effectively, they were broadening the experience of their network by exposing it to the sort of variations that are found in real handwriting.

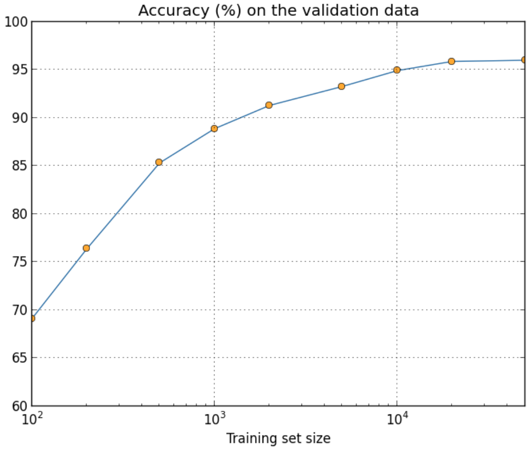
Variations on this idea can be used to improve performance on many learning tasks, not just handwriting recognition. The general principle is to expand the training data by applying operations that reflect real-world variation. It's not difficult to think of ways of doing this. Suppose, for example, that you're building a neural network to do speech recognition. We humans can recognize speech even in the presence of distortions such as background noise. And so you can expand your data by adding background noise. We can also recognize speech if it's sped up or slowed down. So that's another way we can expand the training data. These techniques are not always used - for instance, instead of expanding the training data by adding noise, it may well be more efficient to clean up the input to the network by first applying a noise reduction filter. Still, it's worth keeping the idea of expanding the training data in mind, and looking for opportunities to apply it.

Exercise

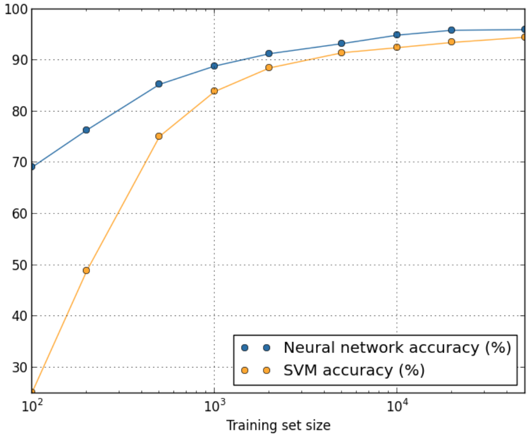
- As discussed above, one way of expanding the MNIST training data is to use small rotations of training images. What's a problem that might occur if we allow arbitrarily large rotations of training images?

[19] Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis, by Patrice Simard, Dave Steinkraus, and John Platt (2003).

An aside on big data and what it means to compare classification accuracies: Let's look again at how our neural network's accuracy varies with training set size:



Suppose that instead of using a neural network we use some other machine learning technique to classify digits. For instance, let's try using the support vector machines (SVM) which we met briefly back in Chapter 1. As was the case in Chapter 1, don't worry if you're not familiar with SVMs, we don't need to understand their details. Instead, we'll use the SVM supplied by the scikit-learn library. Here's how SVM performance varies as a function of training set size. I've plotted the neural net results as well, to make comparison easy[20]:



Probably the first thing that strikes you about this graph is that our neural network outperforms the SVM for every training set size. That's nice, although you shouldn't read too much into it, since I just used the out-of-the-box settings from scikit-learn's SVM, while we've done a fair bit of work improving our neural network. A more subtle but more interesting fact about the graph is that if we train our SVM using 50,000 images then it actually has better

[20] This graph was produced with the program `more_data.py` (as were the last few graphs).

performance (94.48 percent accuracy) than our neural network does when trained using 5,000 images (93.24 percent accuracy). In other words, more training data can sometimes compensate for differences in the machine learning algorithm used.

Something even more interesting can occur. Suppose we're trying to solve a problem using two machine learning algorithms, algorithm A and algorithm B. It sometimes happens that algorithm A will outperform algorithm B with one set of training data, while algorithm B will outperform algorithm A with a different set of training data. We don't see that above – it would require the two graphs to cross – but it does happen[21]. The correct response to the question "Is algorithm A better than algorithm B?" is really: "What training data set are you using?"

All this is a caution to keep in mind, both when doing development, and when reading research papers. Many papers focus on finding new tricks to wring out improved performance on standard benchmark data sets. "Our whiz-bang technique gave us an improvement of X percent on standard benchmark Y" is a canonical form of research claim. Such claims are often genuinely interesting, but they must be understood as applying only in the context of the specific training data set used. Imagine an alternate history in which the people who originally created the benchmark data set had a larger research grant. They might have used the extra money to collect more training data. It's entirely possible that the "improvement" due to the whiz-bang technique would disappear on a larger data set. In other words, the purported improvement might be just an accident of history. The message to take away, especially in practical applications, is that what we want is both better algorithms and better training data. It's fine to look for better algorithms, but make sure you're not focusing on better algorithms to the exclusion of easy wins getting more or better training data.

Problem

- **(Research problem)** How do our machine learning algorithms perform in the limit of very large data sets? For any given algorithm it's natural to attempt to define a notion of asymptotic performance in the limit of truly big data. A quick-and-dirty approach to this problem is to simply try fitting curves to graphs like those shown above, and then to extrapolate the fitted curves out to infinity. An objection to this approach is that different approaches to curve fitting will give different notions of asymptotic performance. Can you find a principled justification for fitting to some particular class of curves? If so, compare the asymptotic performance of several different machine learning algorithms.

Summing up: We've now completed our dive into overfitting and regularization. Of course, we'll return again to the issue. As I've mentioned several times, overfitting is a major problem in neural networks, especially as computers get more powerful, and we have the ability to train larger networks. As a result there's a pressing need to develop powerful regularization techniques to reduce overfitting, and this is an extremely active area of current work.

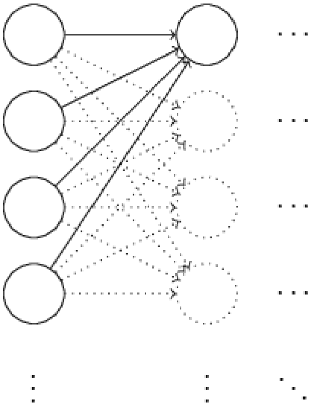
3.3 Weight initialization

When we create our neural networks, we have to make choices for the initial weights and biases. Up to now, we've been choosing them according to a prescription which I discussed only briefly back in Chapter 1. Just to remind you, that prescription was to choose both the weights and biases using independent Gaussian random variables, normalized to have mean 0 and standard deviation 1. While this approach has worked well, it was quite ad hoc, and

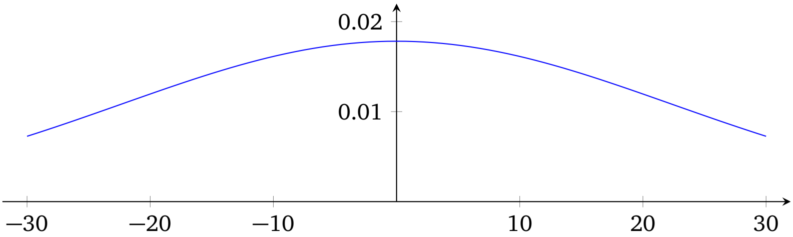
[21] Striking examples may be found in [link]*Scaling to very very large corpora for natural language disambiguation*, by Michele Banko and Eric Brill (2001).

it's worth revisiting to see if we can find a better way of setting our initial weights and biases, and perhaps help our neural networks learn faster.

It turns out that we can do quite a bit better than initializing with normalized Gaussians. To see why, suppose we're working with a network with a large number – say 1,000 – of input neurons. And let's suppose we've used normalized Gaussians to initialize the weights connecting to the first hidden layer. For now I'm going to concentrate specifically on the weights connecting the input neurons to the first neuron in the hidden layer, and ignore the rest of the network:



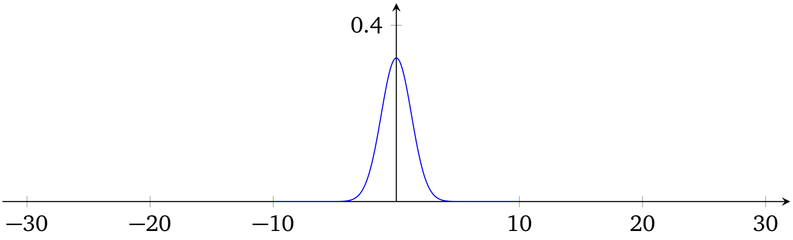
We'll suppose for simplicity that we're trying to train using a training input x in which half the input neurons are on, i.e., set to 1, and half the input neurons are off, i.e., set to 0. The argument which follows applies more generally, but you'll get the gist from this special case. Let's consider the weighted sum $z = \sum_j w_j x_j + b$ of inputs to our hidden neuron. 500 terms in this sum vanish, because the corresponding input x_j is zero. And so z is a sum over a total of 501 normalized Gaussian random variables, accounting for the 500 weight terms and the 1 extra bias term. Thus z is itself distributed as a Gaussian with mean zero and standard deviation $\sqrt{501} \approx 22.4$. That is, z has a very broad Gaussian distribution, not sharply peaked at all:



In particular, we can see from this graph that it's quite likely that $|z|$ will be pretty large, i.e., either $z \gg 1$ or $z \ll 1$. If that's the case then the output $\sigma(z)$ from the hidden neuron will be very close to either 1 or 0. That means our hidden neuron will have saturated. And when that happens, as we know, making small changes in the weights will make only absolutely miniscule changes in the activation of our hidden neuron. That miniscule change in the activation of the hidden neuron will, in turn, barely affect the rest of the neurons in the network at all, and we'll see a correspondingly miniscule change in the cost function. As a result, those weights will only learn very slowly when we use the gradient descent algorithm[22]. It's similar to the problem we discussed earlier in this chapter, in which output neurons which saturated on the wrong value caused learning to slow down. We addressed that earlier problem with a clever choice of cost function. Unfortunately, while that helped with saturated output neurons, it does nothing at all for the problem with saturated hidden neurons.

I've been talking about the weights input to the first hidden layer. Of course, similar arguments apply also to later hidden layers: if the weights in later hidden layers are initialized using normalized Gaussians, then activations will often be very close to 0 or 1, and learning will proceed very slowly.

Is there some way we can choose better initializations for the weights and biases, so that we don't get this kind of saturation, and so avoid a learning slowdown? Suppose we have a neuron with $_{\rm in}$ input weights. Then we shall initialize those weights as Gaussian random variables with mean 0 and standard deviation $1/\sqrt{n_{\rm in}}$. That is, we'll squash the Gaussians down, making it less likely that our neuron will saturate. We'll continue to choose the bias as a Gaussian with mean 0 and standard deviation 1, for reasons I'll return to in a moment. With these choices, the weighted sum $z = \sum_j w_j x_j + b$ will again be a Gaussian random variable with mean 0, but it'll be much more sharply peaked than it was before. Suppose, as we did earlier, that 500 of the inputs are zero and 500 are 1. Then it's easy to show (see the exercise below) that z has a Gaussian distribution with mean 0 and standard deviation $\sqrt{3/2} = 1.22\ldots$. This is much more sharply peaked than before, so much so that even the graph below understates the situation, since I've had to rescale the vertical axis, when compared to the earlier graph:



Such a neuron is much less likely to saturate, and correspondingly much less likely to have problems with a learning slowdown.

Exercise

- Verify that the standard deviation of $z = \sum_j w_j x_j + b$ in the paragraph above is $\sqrt{3/2}$. It may help to know that: (a) the variance of a sum of independent random variables is the sum of the variances of the individual random variables; and (b) the variance is the square of the standard deviation.

I stated above that we'll continue to initialize the biases as before, as Gaussian random variables with a mean of 0 and a standard deviation of 1. This is okay, because it doesn't make it too much more likely that our neurons will saturate. In fact, it doesn't much matter how we initialize the biases, provided we avoid the problem with saturation. Some people go so far as to initialize all the biases to 0, and rely on gradient descent to learn appropriate biases. But since it's unlikely to make much difference, we'll continue with the same initialization procedure as before.

Let's compare the results for both our old and new approaches to weight initialization, using the MNIST digit classification task. As before, we'll use 30 hidden neurons, a mini-batch size of 10, a regularization parameter $\lambda = 5.0$, and the cross-entropy cost function. We will decrease the learning rate slightly from $\eta = 0.5$ to 0.1, since that makes the results a little

[22] We discussed this in more detail in Chapter 2, where we used the equations of backpropagation to show that weights input to saturated neurons learned slowly.

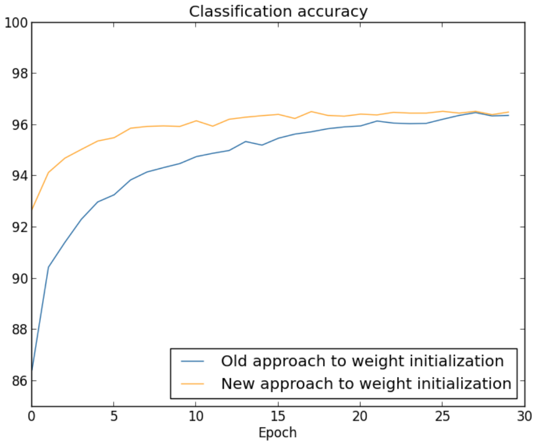
more easily visible in the graphs. We can train using the old method of weight initialization:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.1, lmbda = 5.0, evaluation_data=validation_data,
... monitor_evaluation_accuracy=True)
```

We can also train using the new approach to weight initialization. This is actually even easier, since network2's default way of initializing the weights is using this new approach. That means we can omit the `net.large_weight_initializer()` call above:

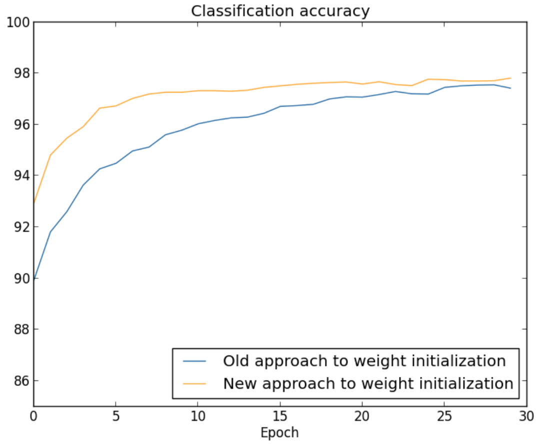
```
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.SGD(training_data, 30, 10, 0.1, lmbda = 5.0, evaluation_data=validation_data,
... monitor_evaluation_accuracy=True)
```

Plotting the results[23], we obtain:



In both cases, we end up with a classification accuracy somewhat over 96 percent. The final classification accuracy is almost exactly the same in the two cases. But the new initialization technique brings us there much, much faster. At the end of the first epoch of training the old approach to weight initialization has a classification accuracy under 87 percent, while the new approach is already almost 93 percent. What appears to be going on is that our new approach to weight initialization starts us off in a much better regime, which lets us get good results much more quickly. The same phenomenon is also seen if we plot results with 100 hidden neurons:

[23] The program used to generate this and the next graph is `weight_initialization.py`.



In this case, the two curves don't quite meet. However, my experiments suggest that with just a few more epochs of training (not shown) the accuracies become almost exactly the same. So on the basis of these experiments it looks as though the improved weight initialization only speeds up learning, it doesn't change the final performance of our networks. However, in Chapter 4 we'll see examples of neural networks where the long-run behaviour is significantly better with the $1/\sqrt{n_{\rm in}}$ weight initialization. Thus it's not only the speed of learning which is improved, it's sometimes also the final performance.

The $1/\sqrt{n_{\rm in}}$ approach to weight initialization helps improve the way our neural nets learn. Other techniques for weight initialization have also been proposed, many building on this basic idea. I won't review the other approaches here, since $1/\sqrt{n_{\rm in}}$ works well enough for our purposes. If you're interested in looking further, I recommend looking at the discussion on pages 14 and 15 of a 2012 paper by Yoshua Bengio[24], as well as the references therein.

Problem

- **Connecting regularization and the improved method of weight initialization** L2 regularization sometimes automatically gives us something similar to the new approach to weight initialization. Suppose we are using the old approach to weight initialization. Sketch a heuristic argument that: (1) supposing λ is not too small, the first epochs of training will be dominated almost entirely by weight decay; (2) provided $\eta\lambda \ll n$ the weights will decay by a factor of $\exp(-\eta\lambda/m)$ per epoch; and (3) supposing λ is not too large, the weight decay will tail off when the weights are down to a size around $1/\sqrt{n_{\rm in}}$, where n is the total number of weights in the network. Argue that these conditions are all satisfied in the examples graphed in this section.

3.4 Handwriting recognition revisited: the code

Let's implement the ideas we've discussed in this chapter. We'll develop a new program, `network2.py`, which is an improved version of the program `network.py` we developed in Chapter 1. If you haven't looked at `network.py` in a while then you may find it helpful to spend a few minutes quickly reading over the earlier discussion. It's only 74 lines of code, and is easily understood.

[24] [link]Practical Recommendations for Gradient-Based Training of Deep Architectures, by Yoshua Bengio (2012).

As was the case in network.py, the star of network2.py is the Network class, which we use to represent our neural networks. We initialize an instance of Network with a list of sizes for the respective layers in the network, and a choice for the cost to use, defaulting to the cross-entropy:

```
class Network(object):
  def __init__(self, sizes, cost=CrossEntropyCost):
    self.num_layers = len(sizes)
    self.sizes = sizes
    self.default_weight_initializer()
    self.cost=cost
```

The first couple of lines of the __init__ method are the same as in network.py, and are pretty self-explanatory. But the next two lines are new, and we need to understand what they're doing in detail.

Let's start by examining the default_weight_initializer method. This makes use of our new and improved approach to weight initialization. As we've seen, in that approach the weights input to a neuron are initialized as Gaussian random variables with mean 0 and standard deviation 1 divided by the square root of the number of connections input to the neuron. Also in this method we'll initialize the biases, using Gaussian random variables with mean 0 and standard deviation 1. Here's the code:

```
def default_weight_initializer(self):
  self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
  self.weights = [np.random.randn(y, x)/np.sqrt(x) for x, y in zip(self.sizes[:-1], self.
     sizes[1:])]
```

To understand the code, it may help to recall that np is the Numpy library for doing linear algebra. We'll import Numpy at the beginning of our program. Also, notice that we don't initialize any biases for the first layer of neurons. We avoid doing this because the first layer is an input layer, and so any biases would not be used. We did exactly the same thing in network.py.

Complementing the default_weight_initializer we'll also include a large_weight_initializer method. This method initializes the weights and biases using the old approach from Chapter 1, with both weights and biases initialized as Gaussian random variables with mean 0 and standard deviation 1. The code is, of course, only a tiny bit different from the default_weight_initializer:

```
def large_weight_initializer(self):
  self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
  self.weights = [np.random.randn(y, x) for x, y in zip(self.sizes[:-1], self.sizes[1:])]
```

I've included the large_weight_initializer method mostly as a convenience to make it easier to compare the results in this chapter to those in Chapter 1. I can't think of many practical situations where I would recommend using it!

The second new thing in Network's __init__ method is that we now initialize a cost attribute. To understand how that works, let's look at the class we use to represent the cross-entropy cost[25]:

```
class CrossEntropyCost(object):
  @staticmethod
  def fn(a, y):
    return np.sum(np.nan_to_num(-y*np.log(a)-(1-y)*np.log(1-a)))
  @staticmethod
  def delta(z, a, y):
    return (a-y)
```

[25] If you're not familiar with Python's static methods you can ignore the @staticmethod decorators, and just treat fn and delta as ordinary methods. If you're curious about details, all @staticmethod does is tell the Python interpreter that the method which follows doesn't depend on the object in any way. That's why self isn't passed as a parameter to the fn and delta methods.

Let's break this down. The first thing to observe is that even though the cross-entropy is, mathematically speaking, a function, we've implemented it as a Python class, not a Python function. Why have I made that choice? The reason is that the cost plays two different roles in our network. The obvious role is that it's a measure of how well an output activation, a, matches the desired output, y. This role is captured by the CrossEntropyCost.fn method. (Note, by the way, that the np.nan_to_num call inside CrossEntropyCost.fn ensures that Numpy deals correctly with the log of numbers very close to zero.) But there's also a second way the cost function enters our network. Recall from Chapter 2 that when running the backpropagation algorithm we need to compute the network's output error, δ^L. The form of the output error depends on the choice of cost function: different cost function, different form for the output error. For the cross-entropy the output error is, as we saw in Equation (3.12),

$$\delta^L = a^L - y. \tag{3.44}$$

For this reason we define a second method, CrossEntropyCost.delta, whose purpose is to tell our network how to compute the output error. And then we bundle these two methods up into a single class containing everything our networks need to know about the cost function.

In a similar way, network2.py also contains a class to represent the quadratic cost function. This is included for comparison with the results of Chapter 1, since going forward we'll mostly use the cross entropy. The code is just below. The QuadraticCost.fn method is a straightforward computation of the quadratic cost associated to the actual output, a, and the desired output, y. The value returned by QuadraticCost.delta is based on the expression (30) for the output error for the quadratic cost, which we derived back in Chapter 2.

```
class QuadraticCost(object):
  @staticmethod
  def fn(a, y):
    return 0.5*np.linalg.norm(a-y)**2
  @staticmethod
  def delta(z, a, y):
    return (a-y) * sigmoid_prime(z)
```

We've now understood the main differences between network2.py and network.py. It's all pretty simple stuff. There are a number of smaller changes, which I'll discuss below, including the implementation of L2 regularization. Before getting to that, let's look at the complete code for network2.py. You don't need to read all the code in detail, but it is worth understanding the broad structure, and in particular reading the documentation strings, so you understand what each piece of the program is doing. Of course, you're also welcome to delve as deeply as you wish! If you get lost, you may wish to continue reading the prose below, and return to the code later. Anyway, here's the code:

```
"""network2.py
~~~~~~~~~~~~~~


An improved version of network.py, implementing the stochastic
gradient descent learning algorithm for a feedforward neural network.
Improvements include the addition of the cross-entropy cost function,
regularization, and better initialization of network weights. Note
that I have focused on making the code simple, easily readable, and
easily modifiable. It is not optimized, and omits many desirable
features.
"""
#### Libraries
# Standard library
import json
import random
import sys
```

```
# Third-party libraries
import numpy as np

#### Define the quadratic and cross-entropy cost functions

class QuadraticCost(object):

  @staticmethod
  def fn(a, y):
    """Return the cost associated with an output ``a`` and desired output ``y``.
    """
    return 0.5*np.linalg.norm(a-y)**2

  @staticmethod
  def delta(z, a, y):
    """Return the error delta from the output layer."""
    return (a-y) * sigmoid_prime(z)

class CrossEntropyCost(object):

  @staticmethod
    def fn(a, y):
    """Return the cost associated with an output ``a`` and desired output
    ``y``.  Note that np.nan_to_num is used to ensure numerical
    stability.  In particular, if both ``a`` and ``y`` have a 1.0
    in the same slot, then the expression (1-y)*np.log(1-a)
    returns nan.  The np.nan_to_num ensures that that is converted
    to the correct value (0.0).

    """
    return np.sum(np.nan_to_num(-y*np.log(a)-(1-y)*np.log(1-a)))

  @staticmethod
  def delta(z, a, y):
    """Return the error delta from the output layer.  Note that the
    parameter ``z`` is not used by the method.  It is included in
    the method's parameters in order to make the interface
    consistent with the delta method for other cost classes.

    """
    return (a-y)


#### Main Network class
class Network(object):

  def __init__(self, sizes, cost=CrossEntropyCost):
    """The list ``sizes`` contains the number of neurons in the respective
    layers of the network.  For example, if the list was [2, 3, 1]
    then it would be a three-layer network, with the first layer
    containing 2 neurons, the second layer 3 neurons, and the
    third layer 1 neuron.  The biases and weights for the network
    are initialized randomly, using
    ``self.default_weight_initializer`` (see docstring for that
    method).

    """
    self.num_layers = len(sizes)
    self.sizes = sizes
    self.default_weight_initializer()
    self.cost=cost

  def default_weight_initializer(self):
    """Initialize each weight using a Gaussian distribution with mean 0
    and standard deviation 1 over the square root of the number of
    weights connecting to the same neuron.  Initialize the biases
    using a Gaussian distribution with mean 0 and standard
    deviation 1.

    Note that the first layer is assumed to be an input layer, and
    by convention we won't set any biases for those neurons, since
    biases are only ever used in computing the outputs from later
    layers.
  """
  self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
  self.weights = [np.random.randn(y, x)/np.sqrt(x) for x, y in zip(self.sizes[:-1],
   self.sizes[1:])]

def large_weight_initializer(self):
  """Initialize the weights using a Gaussian distribution with mean 0
  and standard deviation 1.  Initialize the biases using a
  Gaussian distribution with mean 0 and standard deviation 1.

  Note that the first layer is assumed to be an input layer, and
  by convention we won't set any biases for those neurons, since
  biases are only ever used in computing the outputs from later
  layers.

  This weight and bias initializer uses the same approach as in
  Chapter 1, and is included for purposes of comparison.  It
  will usually be better to use the default weight initializer
  instead.

  """
  self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
  self.weights = [np.random.randn(y, x)
  for x, y in zip(self.sizes[:-1], self.sizes[1:])]

def feedforward(self, a):
  """Return the output of the network if ``a`` is input."""
  for b, w in zip(self.biases, self.weights):
  a = sigmoid(np.dot(w, a)+b)
  return a

def SGD(self, training_data, epochs, mini_batch_size, eta, lmbda = 0.0, evaluation_data
   =None, monitor_evaluation_cost=False, monitor_evaluation_accuracy=False,
   monitor_training_cost=False, monitor_training_accuracy=False):
  """Train the neural network using mini-batch stochastic gradient
  descent.  The ``training_data`` is a list of tuples ``(x, y)``
  representing the training inputs and the desired outputs.  The
  other non-optional parameters are self-explanatory, as is the
  regularization parameter ``lmbda``.  The method also accepts
  ``evaluation_data``, usually either the validation or test
  data.  We can monitor the cost and accuracy on either the
  evaluation data or the training data, by setting the
  appropriate flags.  The method returns a tuple containing four
  lists: the (per-epoch) costs on the evaluation data, the
  accuracies on the evaluation data, the costs on the training
  data, and the accuracies on the training data.  All values are
  evaluated at the end of each training epoch.  So, for example,
  if we train for 30 epochs, then the first element of the tuple
  will be a 30-element list containing the cost on the
  evaluation data at the end of each epoch. Note that the lists
  are empty if the corresponding flag is not set.

  """
  if evaluation_data:
    n_data = len(evaluation_data)
  n = len(training_data)
  evaluation_cost, evaluation_accuracy = [], []
  training_cost, training_accuracy = [], []
  for j in xrange(epochs):
    random.shuffle(training_data)
    mini_batches = [
      training_data[k:k+mini_batch_size]
      for k in xrange(0, n, mini_batch_size)]
    for mini_batch in mini_batches:
      self.update_mini_batch(
        mini_batch, eta, lmbda, len(training_data))
    print "Epoch %s training complete" % j
    if monitor_training_cost:
      cost = self.total_cost(training_data, lmbda)
      training_cost.append(cost)
      print "Cost on training data: {}".format(cost)
    if monitor_training_accuracy:
      accuracy = self.accuracy(training_data, convert=True)
      training_accuracy.append(accuracy)
```

```
        print "Accuracy on training data: {} / {}".format(accuracy, n)
      if monitor_evaluation_cost:
        cost = self.total_cost(evaluation_data, lmbda, convert=True)
        evaluation_cost.append(cost)
        print "Cost on evaluation data: {}".format(cost)
      if monitor_evaluation_accuracy:
        accuracy = self.accuracy(evaluation_data)
        evaluation_accuracy.append(accuracy)
        print "Accuracy on evaluation data: {} / {}".format(self.accuracy(evaluation_data
     ), n_data)
      print
    return evaluation_cost, evaluation_accuracy, training_cost, training_accuracy

  def update_mini_batch(self, mini_batch, eta, lmbda, n):
    """Update the network's weights and biases by applying gradient
    descent using backpropagation to a single mini batch.  The
    ``mini_batch`` is a list of tuples ``(x, y)``, ``eta`` is the
    learning rate, ``lmbda`` is the regularization parameter, and
    ``n`` is the total size of the training data set.

    """
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
      delta_nabla_b, delta_nabla_w = self.backprop(x, y)
      nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
      nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [(1-eta*(lmbda/n))*w-(eta/len(mini_batch))*nw
            for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
            for b, nb in zip(self.biases, nabla_b)]

  def backprop(self, x, y):
    """Return a tuple ``(nabla_b, nabla_w)`` representing the
    gradient for the cost function C_x.  ``nabla_b`` and
    ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
    to ``self.biases`` and ``self.weights``."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
      z = np.dot(w, activation)+b
      zs.append(z)
      activation = sigmoid(z)
      activations.append(activation)
    # backward pass
    delta = (self.cost).delta(zs[-1], activations[-1], y)
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    # Note that the variable l in the loop below is used a little
    # differently to the notation in Chapter 2 of the book.  Here,
    # l = 1 means the last layer of neurons, l = 2 is the
    # second-last layer, and so on.  It's a renumbering of the
    # scheme in the book, used here to take advantage of the fact
    # that Python can use negative indices in lists.
    for l in xrange(2, self.num_layers):
      z = zs[-l]
      sp = sigmoid_prime(z)
      delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
      nabla_b[-l] = delta
      nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)

  def accuracy(self, data, convert=False):
    """Return the number of inputs in ``data`` for which the neural
    network outputs the correct result. The neural network's
    output is assumed to be the index of whichever neuron in the
    final layer has the highest activation.

    The flag ``convert`` should be set to False if the data set is
    validation or test data (the usual case), and to True if the
    data set is the training data. The need for this flag arises
    due to differences in the way the results ``y`` are
    represented in the different data sets.  In particular, it
    flags whether we need to convert between the different
    representations.  It may seem strange to use different
    representations for the different data sets.  Why not use the
    same representation for all three data sets?  It's done for
    efficiency reasons -- the program usually evaluates the cost
    on the training data and the accuracy on other data sets.
    These are different types of computations, and using different
    representations speeds things up.  More details on the
    representations can be found in
    mnist_loader.load_data_wrapper.

    """
    if convert:
      results = [(np.argmax(self.feedforward(x)), np.argmax(y)) for (x, y) in data]
    else:
      results = [(np.argmax(self.feedforward(x)), y)  for (x, y) in data]
    return sum(int(x == y) for (x, y) in results)

  def total_cost(self, data, lmbda, convert=False):
    """Return the total cost for the data set ``data``.  The flag
    ``convert`` should be set to False if the data set is the
    training data (the usual case), and to True if the data set is
    the validation or test data.  See comments on the similar (but
    reversed) convention for the ``accuracy`` method, above.
    """
    cost = 0.0
    for x, y in data:
      a = self.feedforward(x)
    if convert:
      y = vectorized_result(y)
    cost += self.cost.fn(a, y)/len(data)
    cost += 0.5*(lmbda/len(data))*sum(np.linalg.norm(w)**2 for w in self.weights)
    return cost

  def save(self, filename):
    """Save the neural network to the file ``filename``."""
    data = {"sizes": self.sizes,
      "weights": [w.tolist() for w in self.weights],
      "biases": [b.tolist() for b in self.biases],
      "cost": str(self.cost.__name__)}
    f = open(filename, "w")
    json.dump(data, f)
    f.close()

  #### Loading a Network
  def load(filename):
    """Load a neural network from the file ``filename``.  Returns an
    instance of Network.

    """
    f = open(filename, "r")
    data = json.load(f)
    f.close()
    cost = getattr(sys.modules[__name__], data["cost"])
    net = Network(data["sizes"], cost=cost)
    net.weights = [np.array(w) for w in data["weights"]]
    net.biases = [np.array(b) for b in data["biases"]]
    return net

  #### Miscellaneous functions
  def vectorized_result(j):
    """Return a 10-dimensional unit vector with a 1.0 in the j'th position
    and zeroes elsewhere.  This is used to convert a digit (0...9)
    into a corresponding desired output from the neural network.

    """
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e

  def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

  def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

One of the more interesting changes in the code is to include L2 regularization. Although this is a major conceptual change, it's so trivial to implement that it's easy to miss in the code. For the most part it just involves passing the parameter lmbda to various methods, notably the Network.SGD method. The real work is done in a single line of the program, the fourth-last line of the Network.update_mini_batch method. That's where we modify the gradient descent update rule to include weight decay. But although the modification is tiny, it has a big impact on results!

This is, by the way, common when implementing new techniques in neural networks. We've spent thousands of words discussing regularization. It's conceptually quite subtle and difficult to understand. And yet it was trivial to add to our program! It occurs surprisingly often that sophisticated techniques can be implemented with small changes to code.

Another small but important change to our code is the addition of several optional flags to the stochastic gradient descent method, Network.SGD. These flags make it possible to monitor the cost and accuracy either on the training_data or on a set of evaluation_data which can be passed to Network.SGD. We've used these flags often earlier in the chapter, but let me give an example of how it works, just to remind you:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.SGD(training_data, 30, 10, 0.5, lmbda = 5.0, evaluation_data=validation_data,
... monitor_evaluation_accuracy=True, monitor_evaluation_cost=True,
      monitor_training_accuracy=True,
... monitor_training_cost=True)
```

Here, we're setting the evaluation_data to be the validation_data. But we could also have monitored performance on the test_data or any other data set. We also have four flags telling us to monitor the cost and accuracy on both the evaluation_data and the training_data. Those flags are False by default, but they've been turned on here in order to monitor our Network's performance. Furthermore, network2.py's Network.SGD method returns a four-element tuple representing the results of the monitoring. We can use this as follows:

```
>>> evaluation_cost, evaluation_accuracy, training_cost, training_accuracy = net.SGD(
      training_data, 30, 10, 0.5, lmbda = 5.0, evaluation_data=validation_data,
      monitor_evaluation_accuracy=True, monitor_evaluation_cost=True,
      monitor_training_accuracy=True, monitor_training_cost=True)
```

So, for example, evaluation_cost will be a 30-element list containing the cost on the evaluation data at the end of each epoch. This sort of information is extremely useful in understanding a network's behaviour. It can, for example, be used to draw graphs showing how the network learns over time. Indeed, that's exactly how I constructed all the graphs earlier in the chapter. Note, however, that if any of the monitoring flags are not set, then the corresponding element in the tuple will be the empty list.

Other additions to the code include a Network.save method, to save Network objects to disk, and a function to load them back in again later. Note that the saving and loading is done using JSON, not Python's pickle or cPickle modules, which are the usual way we save and load objects to and from disk in Python. Using JSON requires more code than pickle or cPickle would. To understand why I've used JSON, imagine that at some time in the future we decided to change our Network class to allow neurons other than sigmoid neurons. To implement that change we'd most likely change the attributes defined in the `Network.__init__` method. If we've simply pickled the objects that would cause our load function to fail. Using JSON to do the serialization explicitly makes it easy to ensure that old Networks will still load.

There are many other minor changes in the code for `network2.py`, but they're all simple variations on `network.py`. The net result is to expand our 74-line program to a far more capable 152 lines.

Problems

- Modify the code above to implement L1 regularization, and use L1 regularization to classify MNIST digits using a 30 hidden neuron network. Can you find a regularization parameter that enables you to do better than running unregularized?
- Take a look at the `Network.cost_derivative` method in `network.py`. That method was written for the quadratic cost. How would you rewrite the method for the cross-entropy cost? Can you think of a problem that might arise in the cross-entropy version? In `network2.py` we've eliminated the `Network.cost_derivative` method entirely, instead incorporating its functionality into the `CrossEntropyCost.delta` method. How does this solve the problem you've just identified?

3.5 How to choose a neural network's hyper-parameters?

Up until now I haven't explained how I've been choosing values for hyper-parameters such as the learning rate, η, the regularization parameter, λ, and so on. I've just been supplying values which work pretty well. In practice, when you're using neural nets to attack a problem, it can be difficult to find good hyper-parameters. Imagine, for example, that we've just been introduced to the MNIST problem, and have begun working on it, knowing nothing at all about what hyper-parameters to use. Let's suppose that by good fortune in our first experiments we choose many of the hyper-parameters in the same way as was done earlier this chapter: 30 hidden neurons, a mini-batch size of 10, training for 30 epochs using the cross-entropy. But we choose a learning rate $\eta = 10.0$ and regularization parameter $\lambda = 1000.0$. Here's what I saw on one such run:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 10.0, lmbda = 1000.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)


Epoch 0 training complete
Accuracy on evaluation data: 1030 / 10000

Epoch 1 training complete
Accuracy on evaluation data: 990 / 10000

Epoch 2 training complete
Accuracy on evaluation data: 1009 / 10000

...

Epoch 27 training complete
Accuracy on evaluation data: 1009 / 10000

Epoch 28 training complete
Accuracy on evaluation data: 983 / 10000
```

```
Epoch 29 training complete
Accuracy on evaluation data: 967 / 10000
```

Our classification accuracies are no better than chance! Our network is acting as a random noise generator!

"Well, that's easy to fix," you might say, "just decrease the learning rate and regularization hyper-parameters". Unfortunately, you don't a priori know those are the hyper-parameters you need to adjust. Maybe the real problem is that our 30 hidden neuron network will never work well, no matter how the other hyper-parameters are chosen? Maybe we really need at least 100 hidden neurons? Or 300 hidden neurons? Or multiple hidden layers? Or a different approach to encoding the output? Maybe our network is learning, but we need to train for more epochs? Maybe the mini-batches are too small? Maybe we'd do better switching back to the quadratic cost function? Maybe we need to try a different approach to weight initialization? And so on, on and on and on. It's easy to feel lost in hyper-parameter space. This can be particularly frustrating if your network is very large, or uses a lot of training data, since you may train for hours or days or weeks, only to get no result. If the situation persists, it damages your confidence. Maybe neural networks are the wrong approach to your problem? Maybe you should quit your job and take up beekeeping?

In this section I explain some heuristics which can be used to set the hyper-parameters in a neural network. The goal is to help you develop a workflow that enables you to do a pretty good job setting hyper-parameters. Of course, I won't cover everything about hyper-parameter optimization. That's a huge subject, and it's not, in any case, a problem that is ever completely solved, nor is there universal agreement amongst practitioners on the right strategies to use. There's always one more trick you can try to eke out a bit more performance from your network. But the heuristics in this section should get you started.

Broad strategy: When using neural networks to attack a new problem the first challenge is to get any non-trivial learning, i.e., for the network to achieve results better than chance. This can be surprisingly difficult, especially when confronting a new class of problem. Let's look at some strategies you can use if you're having this kind of trouble.

Suppose, for example, that you're attacking MNIST for the first time. You start out enthusiastic, but are a little discouraged when your first network fails completely, as in the example above. The way to go is to strip the problem down. Get rid of all the training and validation images except images which are 0s or 1s. Then try to train a network to distinguish 0s from 1s. Not only is that an inherently easier problem than distinguishing all ten digits, it also reduces the amount of training data by 80 percent, speeding up training by a factor of 5. That enables much more rapid experimentation, and so gives you more rapid insight into how to build a good network.

You can further speed up experimentation by stripping your network down to the simplest network likely to do meaningful learning. If you believe a `[784, 10]` network can likely do better-than-chance classification of MNIST digits, then begin your experimentation with such a network. It'll be much faster than training a `[784, 30, 10]` network, and you can build back up to the latter.

You can get another speed up in experimentation by increasing the frequency of monitoring. In `network2.py` we monitor performance at the end of each training epoch. With 50,000 images per epoch, that means waiting a little while - about ten seconds per epoch, on my laptop, when training a [784, 30, 10] network - before getting feedback on how well the network is learning. Of course, ten seconds isn't very long, but if you want to trial dozens of hyper-parameter choices it's annoying, and if you want to trial hundreds or thousands of choices it starts to get debilitating. We can get feedback more quickly by monitoring the validation accuracy more often, say, after every 1,000 training images. Furthermore, instead of using the full 10,000 image validation set to monitor performance, we can get a much faster estimate using just 100 validation images. All that matters is that the network sees enough images to do real learning, and to get a pretty good rough estimate of performance. Of course, our program `network2.py` doesn't currently do this kind of monitoring. But as a kludge to achieve a similar effect for the purposes of illustration, we'll strip down our training data to just the first 1,000 MNIST training images. Let's try it and see what happens. (To keep the code below simple I haven't implemented the idea of using only 0 and 1 images. Of course, that can be done with just a little more work.)

```
>>> net = network2.Network([784, 10])
>>> net.SGD(training_data[:1000], 30, 10, 10.0, lmbda = 1000.0, \
... evaluation_data=validation_data[:100], \
... monitor_evaluation_accuracy=True)
Epoch 0 training complete
Accuracy on evaluation data: 10 / 100

Epoch 1 training complete
Accuracy on evaluation data: 10 / 100

Epoch 2 training complete
Accuracy on evaluation data: 10 / 100
...
```

We're still getting pure noise! But there's a big win: we're now getting feedback in a fraction of a second, rather than once every ten seconds or so. That means you can more quickly experiment with other choices of hyper-parameter, or even conduct experiments trialling many different choices of hyper-parameter nearly simultaneously.

In the above example I left λ as $\lambda = 1000.0$, as we used earlier. But since we changed the number of training examples we should really change λ to keep the weight decay the same. That means changing λ to 20.0. If we do that then this is what happens:

```
>>> net = network2.Network([784, 10])
>>> net.SGD(training_data[:1000], 30, 10, 10.0, lmbda = 20.0, \
... evaluation_data=validation_data[:100], \
... monitor_evaluation_accuracy=True)
Epoch 0 training complete
Accuracy on evaluation data: 12 / 100

Epoch 1 training complete
Accuracy on evaluation data: 14 / 100

Epoch 2 training complete
Accuracy on evaluation data: 25 / 100

Epoch 3 training complete
Accuracy on evaluation data: 18 / 100
...
```

Ahah! We have a signal. Not a terribly good signal, but a signal nonetheless. That's something we can build on, modifying the hyper-parameters to try to get further improvement. Maybe we guess that our learning rate needs to be higher. (As you perhaps realize, that's a silly guess, for reasons we'll discuss shortly, but please bear with me.) So to test our guess we try dialing η up to 100.0:

```
>>> net = network2.Network([784, 10])
>>> net.SGD(training_data[:1000], 30, 10, 100.0, lmbda = 20.0, \
... evaluation_data=validation_data[:100], \
... monitor_evaluation_accuracy=True)
Epoch 0 training complete
Accuracy on evaluation data: 10 / 100

Epoch 1 training complete
```

```
Accuracy on evaluation data: 10 / 100

Epoch 2 training complete
Accuracy on evaluation data: 10 / 100

Epoch 3 training complete
Accuracy on evaluation data: 10 / 100
...
```

That's no good! It suggests that our guess was wrong, and the problem wasn't that the learning rate was too low. So instead we try dialing η down to $\eta = 1.0$:

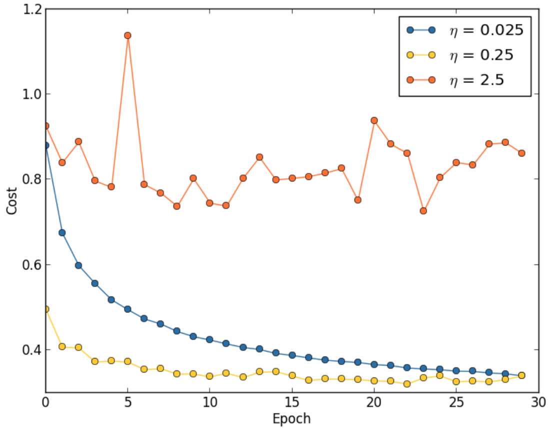
```
>>> net = network2.Network([784, 10])
>>> net.SGD(training_data[:1000], 30, 10, 1.0, lmbda = 20.0, \
... evaluation_data=validation_data[:100], \
... monitor_evaluation_accuracy=True)
Epoch 0 training complete
Accuracy on evaluation data: 62 / 100

Epoch 1 training complete
Accuracy on evaluation data: 42 / 100

Epoch 2 training complete
Accuracy on evaluation data: 43 / 100

Epoch 3 training complete
Accuracy on evaluation data: 61 / 100
...
```

That's better! And so we can continue, individually adjusting each hyper-parameter, gradually improving performance. Once we've explored to find an improved value for η, then we move on to find a good value for λ. Then experiment with a more complex architecture, say a network with 10 hidden neurons. Then adjust the values for η and λ again. Then increase to 20 hidden neurons. And then adjust other hyper-parameters some more. And so on, at each stage evaluating performance using our held-out validation data, and using those evaluations to find better and better hyper-parameters. As we do so, it typically takes longer to witness the impact due to modifications of the hyper-parameters, and so we can gradually decrease the frequency of monitoring.

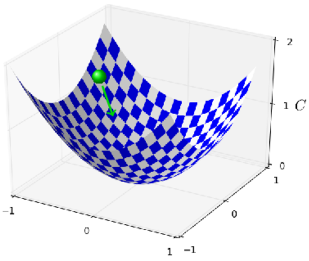
This all looks very promising as a broad strategy. However, I want to return to that initial stage of finding hyper-parameters that enable a network to learn anything at all. In fact, even the above discussion conveys too positive an outlook. It can be immensely frustrating to work with a network that's learning nothing. You can tweak hyper-parameters for days, and still get no meaningful response. And so I'd like to re-emphasize that during the early stages you should make sure you can get quick feedback from experiments. Intuitively, it may seem as though simplifying the problem and the architecture will merely slow you down. In fact, it speeds things up, since you much more quickly find a network with a meaningful signal. Once you've got such a signal, you can often get rapid improvements by tweaking the hyper-parameters. As with many things in life, getting started can be the hardest thing to do.

Okay, that's the broad strategy. Let's now look at some specific recommendations for setting hyper-parameters. I will focus on the learning rate, η, the L2 regularization parameter, λ, and the mini-batch size. However, many of the remarks apply also to other hyper-parameters, including those associated to network architecture, other forms of regularization, and some hyper-parameters we'll meet later in the book, such as the momentum co-efficient.

Learning rate: Suppose we run three MNIST networks with three different learning rates, $\eta = 0.025$, $\eta = 0.25$ and $\eta = 2.5$, respectively. We'll set the other hyper-parameters as for the experiments in earlier sections, running over 30 epochs, with a mini-batch size of 10, and with $\lambda = 5.0$. We'll also return to using the full 50,000 training images. Here's a graph showing the behaviour of the training cost as we train[26]:



With $\eta = 0.025$ the cost decreases smoothly until the final epoch. With $\eta = 0.25$ the cost initially decreases, but after about 20 epochs it is near saturation, and thereafter most of the changes are merely small and apparently random oscillations. Finally, with $\eta = 2.5$ the cost makes large oscillations right from the start. To understand the reason for the oscillations, recall that stochastic gradient descent is supposed to step us gradually down into a valley of the cost function,



However, if η is too large then the steps will be so large that they may actually overshoot the minimum, causing the algorithm to climb up out of the valley instead. That's likely* *This picture is helpful, but it's intended as an intuition-building illustration of what may go on, not as a complete, exhaustive explanation. Briefly, a more complete explanation is as follows: gradient descent uses a first-order approximation to the cost function as a guide to how to decrease the cost. For large η, higher-order terms in the cost function become more important, and may dominate the behaviour, causing gradient descent to break down. This is especially likely as we approach minima and quasi-minima of the cost function, since near such points the gradient becomes small, making it easier for higher-order terms to dominate behaviour. what's causing the cost to oscillate when $\eta = 2.5$. When we choose η=0.25 the initial steps do take us toward a minimum of the cost function, and it's only once we get near

[26] The graph was generated by `multiple_eta.py`.

that minimum that we start to suffer from the overshooting problem. And when we choose $\eta = 0.025$ we don't suffer from this problem at all during the first 30 epochs. Of course, choosing η so small creates another problem, namely, that it slows down stochastic gradient descent. An even better approach would be to start with $\eta = 0.25$, train for 20 epochs, and then switch to $\eta = 0.025$. We'll discuss such variable learning rate schedules later. For now, though, let's stick to figuring out how to find a single good value for the learning rate, η.

With this picture in mind, we can set η as follows. First, we estimate the threshold value for η at which the cost on the training data immediately begins decreasing, instead of oscillating or increasing. This estimate doesn't need to be too accurate. You can estimate the order of magnitude by starting with η=0.01. If the cost decreases during the first few epochs, then you should successively try $\eta = 0.1, 1.0, \ldots$ until you find a value for η where the cost oscillates or increases during the first few epochs. Alternately, if the cost oscillates or increases during the first few epochs when η=0.01, then try η=0.001,0.0001,... until you find a value for η where the cost decreases during the first few epochs. Following this procedure will give us an order of magnitude estimate for the threshold value of η. You may optionally refine your estimate, to pick out the largest value of η at which the cost decreases during the first few epochs, say η=0.5 or η=0.2 (there's no need for this to be super-accurate). This gives us an estimate for the threshold value of η.

Obviously, the actual value of η that you use should be no larger than the threshold value. In fact, if the value of η is to remain usable over many epochs then you likely want to use a value for η that is smaller, say, a factor of two below the threshold. Such a choice will typically allow you to train for many epochs, without causing too much of a slowdown in learning.

In the case of the MNIST data, following this strategy leads to an estimate of 0.1 for the order of magnitude of the threshold value of η. After some more refinement, we obtain a threshold value η=0.5. Following the prescription above, this suggests using η=0.25 as our value for the learning rate. In fact, I found that using η=0.5 worked well enough over 30 epochs that for the most part I didn't worry about using a lower value of η.

This all seems quite straightforward. However, using the training cost to pick η appears to contradict what I said earlier in this section, namely, that we'd pick hyper-parameters by evaluating performance using our held-out validation data. In fact, we'll use validation accuracy to pick the regularization hyper-parameter, the mini-batch size, and network parameters such as the number of layers and hidden neurons, and so on. Why do things differently for the learning rate? Frankly, this choice is my personal aesthetic preference, and is perhaps somewhat idiosyncratic. The reasoning is that the other hyper-parameters are intended to improve the final classification accuracy on the test set, and so it makes sense to select them on the basis of validation accuracy. However, the learning rate is only incidentally meant to impact the final classification accuracy. Its primary purpose is really to control the step size in gradient descent, and monitoring the training cost is the best way to detect if the step size is too big. With that said, this is a personal aesthetic preference. Early on during learning the training cost usually only decreases if the validation accuracy improves, and so in practice it's unlikely to make much difference which criterion you use.

Use early stopping to determine the number of training epochs: As we discussed earlier in the chapter, early stopping means that at the end of each epoch we should compute the classification accuracy on the validation data. When that stops improving, terminate. This makes setting the number of epochs very simple. In particular, it means that we don't need to worry about explicitly figuring out how the number of epochs depends on the other hyper-parameters. Instead, that's taken care of automatically. Furthermore, early stopping also automatically prevents us from overfitting. This is, of course, a good thing, although in the early stages of experimentation it can be helpful to turn off early stopping, so you can see any signs of overfitting, and use it to inform your approach to regularization.

To implement early stopping we need to say more precisely what it means that the classification accuracy has stopped improving. As we've seen, the accuracy can jump around quite a bit, even when the overall trend is to improve. If we stop the first time the accuracy decreases then we'll almost certainly stop when there are more improvements to be had. A better rule is to terminate if the best classification accuracy doesn't improve for quite some time. Suppose, for example, that we're doing MNIST. Then we might elect to terminate if the classification accuracy hasn't improved during the last ten epochs. This ensures that we don't stop too soon, in response to bad luck in training, but also that we're not waiting around forever for an improvement that never comes.

This no-improvement-in-ten rule is good for initial exploration of MNIST. However, networks can sometimes plateau near a particular classification accuracy for quite some time, only to then begin improving again. If you're trying to get really good performance, the no-improvement-in-ten rule may be too aggressive about stopping. In that case, I suggest using the no-improvement-in-ten rule for initial experimentation, and gradually adopting more lenient rules, as you better understand the way your network trains: no-improvement-in-twenty, no-improvement-in-fifty, and so on. Of course, this introduces a new hyper-parameter to optimize! In practice, however, it's usually easy to set this hyper-parameter to get pretty good results. Similarly, for problems other than MNIST, the no-improvement-in-ten rule may be much too aggressive or not nearly aggressive enough, depending on the details of the problem. However, with a little experimentation it's usually easy to find a pretty good strategy for early stopping.

We haven't used early stopping in our MNIST experiments to date. The reason is that we've been doing a lot of comparisons between different approaches to learning. For such comparisons it's helpful to use the same number of epochs in each case. However, it's well worth modifying `network2.py` to implement early stopping:

Problem

- Modify `network2.py` so that it implements early stopping using a no-improvement-in-n epochs strategy, where n is a parameter that can be set.
- Can you think of a rule for early stopping other than no-improvement-in-n? Ideally, the rule should compromise between getting high validation accuracies and not training too long. Add your rule to `network2.py`, and run three experiments comparing the validation accuracies and number of epochs of training to no-improvement-in-10.

Learning rate schedule: We've been holding the learning rate η constant. However, it's often advantageous to vary the learning rate. Early on during the learning process it's likely that the weights are badly wrong. And so it's best to use a large learning rate that causes the weights to change quickly. Later, we can reduce the learning rate as we make more fine-tuned adjustments to our weights.

How should we set our learning rate schedule? Many approaches are possible. One natural approach is to use the same basic idea as early stopping. The idea is to hold the learning rate constant until the validation accuracy starts to get worse. Then decrease the learning rate by some amount, say a factor of two or ten. We repeat this many times, until, say, the learning rate is a factor of 1,024 (or 1,000) times lower than the initial value. Then we terminate.

A variable learning schedule can improve performance, but it also opens up a world of possible choices for the learning schedule. Those choices can be a headache – you can spend forever trying to optimize your learning schedule. For first experiments my suggestion is to use a single, constant value for the learning rate. That'll get you a good first approximation.

Later, if you want to obtain the best performance from your network, it's worth experimenting with a learning schedule, along the lines I've described[27].

Exercise

- Modify `network2.py` so that it implements a learning schedule that: halves the learning rate each time the validation accuracy satisfies the no-improvement-in-10 rule; and terminates when the learning rate has dropped to 1/128 of its original value.

The regularization parameter, λ: I suggest starting initially with no regularization ($\lambda = 0.0$), and determining a value for η, as above. Using that choice of η, we can then use the validation data to select a good value for λ. Start by trialling $\lambda = 1.0$[28], and then increase or decrease by factors of 10, as needed to improve performance on the validation data. Once you've found a good order of magnitude, you can fine tune your value of λ. That done, you should return and re-optimize η again.

Exercise

- It's tempting to use gradient descent to try to learn good values for hyper-parameters such as λ and η. Can you think of an obstacle to using gradient descent to determine λ? Can you think of an obstacle to using gradient descent to determine η?

How I selected hyper-parameters earlier in this book: If you use the recommendations in this section you'll find that you get values for η and λ which don't always exactly match the values I've used earlier in the book. The reason is that the book has narrative constraints that have sometimes made it impractical to optimize the hyper-parameters. Think of all the comparisons we've made of different approaches to learning, e.g., comparing the quadratic and cross-entropy cost functions, comparing the old and new methods of weight initialization, running with and without regularization, and so on. To make such comparisons meaningful, I've usually tried to keep hyper-parameters constant across the approaches being compared (or to scale them in an appropriate way). Of course, there's no reason for the same hyper-parameters to be optimal for all the different approaches to learning, so the hyper-parameters I've used are something of a compromise.

As an alternative to this compromise, I could have tried to optimize the heck out of the hyper-parameters for every single approach to learning. In principle that'd be a better, fairer approach, since then we'd see the best from every approach to learning. However, we've made dozens of comparisons along these lines, and in practice I found it too computationally expensive. That's why I've adopted the compromise of using pretty good (but not necessarily optimal) choices for the hyper-parameters.

Mini-batch size: How should we set the mini-batch size? To answer this question, let's first suppose that we're doing online learning, i.e., that we're using a mini-batch size of 1.

The obvious worry about online learning is that using mini-batches which contain just a single training example will cause significant errors in our estimate of the gradient. In fact, though, the errors turn out to not be such a problem. The reason is that the individual gradient estimates don't need to be super-accurate. All we need is an estimate accurate enough that our cost function tends to keep decreasing. It's as though you are trying to get to the North Magnetic Pole, but have a wonky compass that's 10-20 degrees off each time you look at it. Provided you stop to check the compass frequently, and the compass gets the direction right on average, you'll end up at the North Magnetic Pole just fine.

Based on this argument, it sounds as though we should use online learning. In fact, the situation turns out to be more complicated than that. In a problem [link]in the last chapter I

[27] A readable recent paper which demonstrates the benefits of variable learning rates in attacking MNIST is Deep, Big, Simple Neural Nets Excel on Handwritten Digit Recognition, by Dan Claudiu CireÈŹan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber (2010).

[28] I don't have a good principled justification for using this as a starting value. If anyone knows of a good principled discussion of where to start with λ, I'd appreciate hearing it (mn@michaelnielsen.org).

pointed out that it’s possible to use matrix techniques to compute the gradient update for all examples in a mini-batch simultaneously, rather than looping over them. Depending on the details of your hardware and linear algebra library this can make it quite a bit faster to compute the gradient estimate for a mini-batch of (for example) size 100, rather than computing the mini-batch gradient estimate by looping over the 100 training examples separately. It might take (say) only 50 times as long, rather than 100 times as long.

Now, at first it seems as though this doesn’t help us that much. With our mini-batch of size 100 the learning rule for the weights looks like:

$$w \rightarrow w' = w - \eta \frac{1}{100} \sum_x \nabla C_x, \tag{3.45}$$

where the sum is over training examples in the mini-batch. This is versus

$$w \rightarrow w' = w - \eta \nabla C_x \tag{3.46}$$

for online learning. Even if it only takes 50 times as long to do the mini-batch update, it still seems likely to be better to do online learning, because we’d be updating so much more frequently. Suppose, however, that in the mini-batch case we increase the learning rate by a factor 100, so the update rule becomes

$$w \rightarrow w' = w - \eta \sum_x \nabla C_x. \tag{3.47}$$

That’s a lot like doing 100 separate instances of online learning with a learning rate of η. But it only takes 50 times as long as doing a single instance of online learning. Of course, it’s not truly the same as 100 instances of online learning, since in the mini-batch the âĹĞCx’s are all evaluated for the same set of weights, as opposed to the cumulative learning that occurs in the online case. Still, it seems distinctly possible that using the larger mini-batch would speed things up.

With these factors in mind, choosing the best mini-batch size is a compromise. Too small, and you don’t get to take full advantage of the benefits of good matrix libraries optimized for fast hardware. Too large and you’re simply not updating your weights often enough. What you need is to choose a compromise value which maximizes the speed of learning. Fortunately, the choice of mini-batch size at which the speed is maximized is relatively independent of the other hyper-parameters (apart from the overall architecture), so you don’t need to have optimized those hyper-parameters in order to find a good mini-batch size. The way to go is therefore to use some acceptable (but not necessarily optimal) values for the other hyper-parameters, and then trial a number of different mini-batch sizes, scaling η as above. Plot the validation accuracy versus time (as in, real elapsed time, not epoch!), and choose whichever mini-batch size gives you the most rapid improvement in performance. With the mini-batch size chosen you can then proceed to optimize the other hyper-parameters.

Of course, as you’ve no doubt realized, I haven’t done this optimization in our work. Indeed, our implementation doesn’t use the faster approach to mini-batch updates at all. I’ve simply used a mini-batch size of 10 without comment or explanation in nearly all examples. Because of this, we could have sped up learning by reducing the mini-batch size. I haven’t done this, in part because I wanted to illustrate the use of mini-batches beyond size 1, and in part because my preliminary experiments suggested the speedup would be rather modest. In practical implementations, however, we would most certainly implement the faster approach to mini-batch updates, and then make an effort to optimize the mini-batch size, in order to maximize our overall speed.

Automated techniques: I've been describing these heuristics as though you're optimizing your hyper-parameters by hand. Hand-optimization is a good way to build up a feel for how neural networks behave. However, and unsurprisingly, a great deal of work has been done on automating the process. A common technique is grid search, which systematically searches through a grid in hyper-parameter space. A review of both the achievements and the limitations of grid search (with suggestions for easily-implemented alternatives) may be found in a 2012 paper[29] by James Bergstra and Yoshua Bengio. Many more sophisticated approaches have also been proposed. I won't review all that work here, but do want to mention a particularly promising 2012 paper which used a Bayesian approach to automatically optimize hyper-parameters[30]. The code from the paper is [link]publicly available, and has been used with some success by other researchers.

Summing up: Following the rules-of-thumb I've described won't give you the absolute best possible results from your neural network. But it will likely give you a good start and a basis for further improvements. In particular, I've discussed the hyper-parameters largely independently. In practice, there are relationships between the hyper-parameters. You may experiment with η, feel that you've got it just right, then start to optimize for λ, only to find that it's messing up your optimization for η. In practice, it helps to bounce backward and forward, gradually closing in good values. Above all, keep in mind that the heuristics I've described are rules of thumb, not rules cast in stone. You should be on the lookout for signs that things aren't working, and be willing to experiment. In particular, this means carefully monitoring your network's behaviour, especially the validation accuracy.

The difficulty of choosing hyper-parameters is exacerbated by the fact that the lore about how to choose hyper-parameters is widely spread, across many research papers and software programs, and often is only available inside the heads of individual practitioners. There are many, many papers setting out (sometimes contradictory) recommendations for how to proceed. However, there are a few particularly useful papers that synthesize and distill out much of this lore. Yoshua Bengio has a 2012 paper[31] that gives some practical recommendations for using backpropagation and gradient descent to train neural networks, including deep neural nets. Bengio discusses many issues in much more detail than I have, including how to do more systematic hyper-parameter searches. Another good paper is a 1998 paper[32] by Yann LeCun, Léon Bottou, Genevieve Orr and Klaus-Robert Müller. Both these papers appear in an extremely useful 2012 book that collects many tricks commonly used in neural nets[33]. The book is expensive, but many of the articles have been placed online by their respective authors with, one presumes, the blessing of the publisher, and may be located using a search engine.

One thing that becomes clear as you read these articles and, especially, as you engage in your own experiments, is that hyper-parameter optimization is not a problem that is ever completely solved. There's always another trick you can try to improve performance. There is a saying common among writers that books are never finished, only abandoned. The same is also true of neural network optimization: the space of hyper-parameters is so large that one never really finishes optimizing, one only abandons the network to posterity. So your goal should be to develop a workflow that enables you to quickly do a pretty good job on

[29] Random search for hyper-parameter optimization, by James Bergstra and Yoshua Bengio (2012).

[30] Practical Bayesian optimization of machine learning algorithms, by Jasper Snoek, Hugo Larochelle, and Ryan Adams.

[31] Practical recommendations for gradient-based training of deep architectures, by Yoshua Bengio (2012).

[32] Efficient BackProp, by Yann LeCun, LÃľon Bottou, Genevieve Orr and Klaus-Robert Müller (1998)

[33] Neural Networks: Tricks of the Trade, edited by Grégoire Montavon, Geneviève Orr, and Klaus-Robert Müller.

the optimization, while leaving you the flexibility to try more detailed optimizations, if that's important.

The challenge of setting hyper-parameters has led some people to complain that neural networks require a lot of work when compared with other machine learning techniques. I've heard many variations on the following complaint: "Yes, a well-tuned neural network may get the best performance on the problem. On the other hand, I can try a random forest [or SVM or ... insert your own favorite technique] and it just works. I don't have time to figure out just the right neural network." Of course, from a practical point of view it's good to have easy-to-apply techniques. This is particularly true when you're just getting started on a problem, and it may not be obvious whether machine learning can help solve the problem at all. On the other hand, if getting optimal performance is important, then you may need to try approaches that require more specialist knowledge. While it would be nice if machine learning were always easy, there is no a priori reason it should be trivially simple.

3.6 Other techniques

Each technique developed in this chapter is valuable to know in its own right, but that's not the only reason I've explained them. The larger point is to familiarize you with some of the problems which can occur in neural networks, and with a style of analysis which can help overcome those problems. In a sense, we've been learning how to think about neural nets. Over the remainder of this chapter I briefly sketch a handful of other techniques. These sketches are less in-depth than the earlier discussions, but should convey some feeling for the diversity of techniques available for use in neural networks.

3.6.1 Variations on stochastic gradient descent

Stochastic gradient descent by backpropagation has served us well in attacking the MNIST digit classification problem. However, there are many other approaches to optimizing the cost function, and sometimes those other approaches offer performance superior to mini-batch stochastic gradient descent. In this section I sketch two such approaches, the Hessian and momentum techniques.

Hessian technique: To begin our discussion it helps to put neural networks aside for a bit. Instead, we're just going to consider the abstract problem of minimizing a cost function C which is a function of many variables, $w = w_1, w_2, \ldots$, so $C = C(w)$. By Taylor's theorem, the cost function can be approximated near a point w by

$$C(w + \Delta w) = C(w) + \sum_j \frac{\partial C}{\partial w_j} \Delta w_j + \frac{1}{2} \sum_{jk} \Delta w_j \frac{\partial^2 C}{\partial w_j \partial w_k} \Delta w_k + \ldots \tag{3.48}$$

We can rewrite this more compactly as

$$C(w + \Delta w) = C(w) + \nabla C \cdot \Delta w + \frac{1}{2} \Delta w^T H \Delta w + \ldots, \tag{3.49}$$

where ∇C is the usual gradient vector, and H is a matrix known as the *Hessian matrix*, whose jk-th entry is $\partial^2 C / \partial w_j \partial w_k$. Suppose we approximate C by discarding the higher-order terms represented by . . . above,

$$C(w+\Delta w) \approx C(w) + \nabla C \cdot \Delta w + \frac{1}{2}\Delta w^T H \Delta w. \tag{3.50}$$

Using calculus we can show that the expression on the right-hand side can be minimized[34] by choosing

$$\Delta w = -H^{-1}\nabla C. \tag{3.51}$$

Provided (3.50) is a good approximate expression for the cost function, then we'd expect that moving from the point w to $w + \Delta w = w - H^{-1}\nabla C$ should significantly decrease the cost function. That suggests a possible algorithm for minimizing the cost:

- Choose a starting point, w.
- Update w to a new point$w' = w - H^{-1}\nabla C$, where the Hessian H and ∇C are computed at w.
- Update w' to a new point $w'' = w'H'^1\nabla'C$, where the Hessian H' and $\nabla'C$ are computed at w'.
- . . .

In practice, (3.50) is only an approximation, and it's better to take smaller steps. We do this by repeatedly changing w by an amount $\Delta w = -\eta H^{-1}\nabla C$, where η is known as the learning rate.

This approach to minimizing a cost function is known as the *Hessian technique* or *Hessian optimization*. There are theoretical and empirical results showing that Hessian methods converge on a minimum in fewer steps than standard gradient descent. In particular, by incorporating information about second-order changes in the cost function it's possible for the Hessian approach to avoid many pathologies that can occur in gradient descent. Furthermore, there are versions of the backpropagation algorithm which can be used to compute the Hessian.

If Hessian optimization is so great, why aren't we using it in our neural networks? Unfortunately, while it has many desirable properties, it has one very undesirable property: it's very difficult to apply in practice. Part of the problem is the sheer size of the Hessian matrix. Suppose you have a neural network with 10^7 weights and biases. Then the corresponding Hessian matrix will contain $10^7 \times 10^7 = 10^{14}$ entries. That's a lot of entries! And that makes computing $H^{-1}\nabla C$ extremely difficult in practice. However, that doesn't mean that it's not useful to understand. In fact, there are many variations on gradient descent which are inspired by Hessian optimization, but which avoid the problem with overly-large matrices. Let's take a look at one such technique, momentum-based gradient descent.

Momentum-based gradient descent: Intuitively, the advantage Hessian optimization has is that it incorporates not just information about the gradient, but also information about how the gradient is changing. Momentum-based gradient descent is based on a similar intuition, but avoids large matrices of second derivatives. To understand the momentum technique, think back to our [link]original picture of gradient descent, in which we considered a ball rolling down into a valley. At the time, we observed that gradient descent is, despite its name, only loosely similar to a ball falling to the bottom of a valley. The momentum technique modifies gradient descent in two ways that make it more similar to the physical

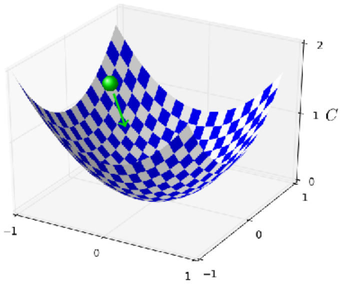
[34] Strictly speaking, for this to be a minimum, and not merely an extremum, we need to assume that the Hessian matrix is positive definite. Intuitively, this means that the function C looks like a valley locally, not a mountain or a saddle.

picture. First, it introduces a notion of "velocity" for the parameters we're trying to optimize. The gradient acts to change the velocity, not (directly) the "position", in much the same way as physical forces change the velocity, and only indirectly affect position. Second, the momentum method introduces a kind of friction term, which tends to gradually reduce the velocity.

Let's give a more precise mathematical description. We introduce velocity variables $v = v_1, v_2, \ldots$, one for each corresponding w_j variable[35]. Then we replace the gradient descent update rule $w \rightarrow w' = w - \eta \nabla C$ by

$$\begin{aligned} v &\rightarrow v' = \mu v - \eta \nabla C \quad (3.52) \\ w &\rightarrow w' = w + v'. \quad (3.53) \end{aligned}$$

In these equations, μ is a hyper-parameter which controls the amount of damping or friction in the system. To understand the meaning of the equations it's helpful to first consider the case where $\mu = 1$, which corresponds to no friction. When that's the case, inspection of the equations shows that the "force" ∇C is now modifying the velocity, vv, and the velocity is controlling the rate of change of w. Intuitively, we build up the velocity by repeatedly adding gradient terms to it. That means that if the gradient is in (roughly) the same direction through several rounds of learning, we can build up quite a bit of steam moving in that direction. Think, for example, of what happens if we're moving straight down a slope:



With each step the velocity gets larger down the slope, so we move more and more quickly to the bottom of the valley. This can enable the momentum technique to work much faster than standard gradient descent. Of course, a problem is that once we reach the bottom of the valley we will overshoot. Or, if the gradient should change rapidly, then we could find ourselves moving in the wrong direction. That's the reason for the μ hyper-parameter in (3.52). I said earlier that μ controls the amount of friction in the system; to be a little more precise, you should think of $1 - \mu$ as the amount of friction in the system. When $\mu = 1$, as we've seen, there is no friction, and the velocity is completely driven by the gradient ∇C. By contrast, when $\mu = 0$ there's a lot of friction, the velocity can't build up, and Equations (3.52) and (3.53) reduce to the usual equation for gradient descent, $w \rightarrow w' = w - \eta \nabla C$. In practice, using a value of μ intermediate between 0 and 1 can give us much of the benefit of being able to build up speed, but without causing overshooting. We can choose such a value for μ using the held-out validation data, in much the same way as we select η and λ.

[35] In a neural net the w_j variables would, of course, include all weights and biases.

I've avoided naming the hyper-parameter μ up to now. The reason is that the standard name for μ is badly chosen: it's called the momentum co-efficient. This is potentially confusing, since μ is not at all the same as the notion of momentum from physics. Rather, it is much more closely related to friction. However, the term momentum co-efficient is widely used, so we will continue to use it.

A nice thing about the momentum technique is that it takes almost no work to modify an implementation of gradient descent to incorporate momentum. We can still use backpropagation to compute the gradients, just as before, and use ideas such as sampling stochastically chosen mini-batches. In this way, we can get some of the advantages of the Hessian technique, using information about how the gradient is changing. But it's done without the disadvantages, and with only minor modifications to our code. In practice, the momentum technique is commonly used, and often speeds up learning. **Exercise**

- What would go wrong if we used $\mu > 1$ in the momentum technique?
- What would go wrong if we used $\mu < 0$ in the momentum technique?

Problem

- Add momentum-based stochastic gradient descent to `network2.py`.

Other approaches to minimizing the cost function: Many other approaches to minimizing the cost function have been developed, and there isn't universal agreement on which is the best approach. As you go deeper into neural networks it's worth digging into the other techniques, understanding how they work, their strengths and weaknesses, and how to apply them in practice. A paper I mentioned earlier[36] introduces and compares several of these techniques, including conjugate gradient descent and the BFGS method (see also the closely related limited-memory BFGS method, known as [link]L-BFGS). Another technique which has recently shown promising results[37] is Nesterov's accelerated gradient technique, which improves on the momentum technique. However, for many problems, plain stochastic gradient descent works well, especially if momentum is used, and so we'll stick to stochastic gradient descent through the remainder of this book.

Other models of artificial neuron

Up to now we've built our neural networks using sigmoid neurons. In principle, a network built from sigmoid neurons can compute any function. In practice, however, networks built using other model neurons sometimes outperform sigmoid networks. Depending on the application, networks based on such alternate models may learn faster, generalize better to test data, or perhaps do both. Let me mention a couple of alternate model neurons, to give you the flavor of some variations in common use.

Perhaps the simplest variation is the tanh (pronounced "tanch") neuron, which replaces the sigmoid function by the hyperbolic tangent function. The output of a tanh neuron with input x, weight vector w, and bias b is given by

$$\tanh(w \cdot x + b), \tag{3.54}$$

where tanh is, of course, the hyperbolic tangent function. It turns out that this is very closely

[36] Efficient BackProp, by Yann LeCun, Léon Bottou, Genevieve Orr and Klaus-Robert Müller (1998).

[37] See, for example, On the importance of initialization and momentum in deep learning, by Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton (2012).

related to the sigmoid neuron. To see this, recall that the tanh function is defined by

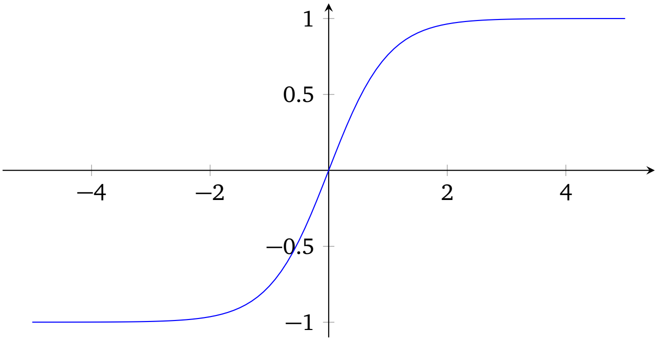
$$\tanh(z) \equiv \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{3.55}$$

With a little algebra it can easily be verified that

$$\sigma(z) = \frac{1 + \tanh(z/2)}{2}, \tag{3.56}$$

that is, tanh is just a rescaled version of the sigmoid function. We can also see graphically that the tanh function has the same shape as the sigmoid function,

tanh function



One difference between tanh neurons and sigmoid neurons is that the output from tanh neurons ranges from −1 to 1, not 0 to 1. This means that if you're going to build a network based on tanh neurons you may need to normalize your outputs (and, depending on the details of the application, possibly your inputs) a little differently than in sigmoid networks.

Similar to sigmoid neurons, a network of tanh neurons can, in principle, compute any function[38] mapping inputs to the range −1 to 1. Furthermore, ideas such as backpropagation and stochastic gradient descent are as easily applied to a network of tanh neurons as to a network of sigmoid neurons.

Exercise

- Prove the identity in Equation (3.56).

Which type of neuron should you use in your networks, the tanh or sigmoid? A priori the answer is not obvious, to put it mildly! However, there are theoretical arguments and some empirical evidence to suggest that the tanh sometimes performs better[39]. Let me briefly give you the flavor of one of the theoretical arguments for tanh neurons. Suppose we're using sigmoid neurons, so all activations in our network are positive. Let's consider the weights w^{l+1}_{jk} input to the j-th neuron in the $l+1$-th layer. The rules for backpropagation (see [link]here) tell us that the associated gradient will be $a^l_k \delta^{l+1}_j$. Because the activations are positive the sign of this gradient will be the same as the sign of δ^{l+1}_j. What this means is that

[38] There are some technical caveats to this statement for both tanh and sigmoid neurons, as well as for the rectified linear neurons discussed below. However, informally it's usually fine to think of neural networks as being able to approximate any function to arbitrary accuracy.

[39] See, for example, [link]Efficient BackProp, by Yann LeCun, Léon Bottou, Genevieve Orr and Klaus-Robert Müller (1998), and [link]Understanding the difficulty of training deep feedforward networks, by Xavier Glorot and Yoshua Bengio (2010).

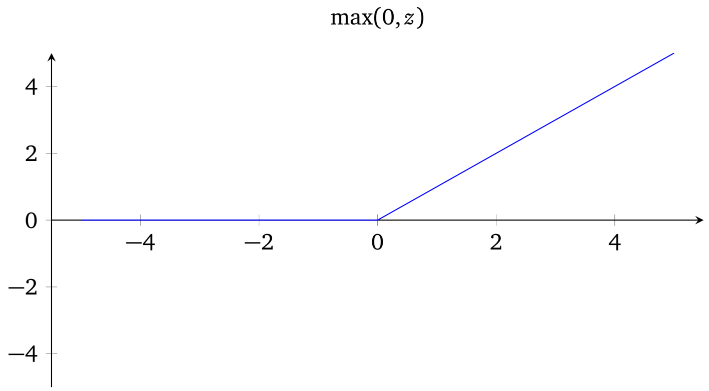
if δ_j^{l+1} is positive then all the weights w_{jk}^{l+1} will decrease during gradient descent, while if δ_j^{l+1} is negative then all the weights w_{jk}^{l+1} will increase during gradient descent. In other words, all weights to the same neuron must either increase together or decrease together. That's a problem, since some of the weights may need to increase while others need to decrease. That can only happen if some of the input activations have different signs. That suggests replacing the sigmoid by an activation function, such as tanh, which allows both positive and negative activations. Indeed, because tanh is symmetric about zero, $\tanh(-z) = -\tanh(z)$, we might even expect that, roughly speaking, the activations in hidden layers would be equally balanced between positive and negative. That would help ensure that there is no systematic bias for the weight updates to be one way or the other.

How seriously should we take this argument? While the argument is suggestive, it's a heuristic, not a rigorous proof that tanh neurons outperform sigmoid neurons. Perhaps there are other properties of the sigmoid neuron which compensate for this problem? Indeed, for many tasks the tanh is found empirically to provide only a small or no improvement in performance over sigmoid neurons. Unfortunately, we don't yet have hard-and-fast rules to know which neuron types will learn fastest, or give the best generalization performance, for any particular application.

Another variation on the sigmoid neuron is the *rectified linear neuron* or *rectified linear* unit. The output of a rectified linear unit with input x, weight vector w, and bias b is given by

$$\max(0, w \cdot x + b). \tag{3.57}$$

Graphically, the rectifying function $\max(0, z)$ looks like this:



Obviously such neurons are quite different from both sigmoid and tanh neurons. However, like the sigmoid and tanh neurons, rectified linear units can be used to compute any function, and they can be trained using ideas such as backpropagation and stochastic gradient descent.

When should you use rectified linear units instead of sigmoid or tanh neurons? Some recent work on image recognition[40] has found considerable benefit in using rectified linear

[40] See, for example, [link]What is the Best Multi-Stage Architecture for Object Recognition?, by Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato and Yann LeCun (2009), [link]Deep Sparse RectiïňĄer Neural Networks, by Xavier Glorot, Antoine Bordes, and Yoshua Bengio (2011), and ImageNet Classification with Deep Convolutional Neural Networks, by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton (2012). Note that these papers fill in important details about how to set up the output layer, cost function, and regularization in networks using rectified linear units. I've glossed over all these details in this brief account. The papers also discuss in more detail the benefits and drawbacks of using rectified linear units. Another informative paper is Rectified Linear Units Improve Restricted

units through much of the network. However, as with tanh neurons, we do not yet have a really deep understanding of when, exactly, rectified linear units are preferable, nor why. To give you the flavor of some of the issues, recall that sigmoid neurons stop learning when they saturate, i.e., when their output is near either 0 or 1. As we've seen repeatedly in this chapter, the problem is that σ' terms reduce the gradient, and that slows down learning. Tanh neurons suffer from a similar problem when they saturate. By contrast, increasing the weighted input to a rectified linear unit will never cause it to saturate, and so there is no corresponding learning slowdown. On the other hand, when the weighted input to a rectified linear unit is negative, the gradient vanishes, and so the neuron stops learning entirely. These are just two of the many issues that make it non-trivial to understand when and why rectified linear units perform better than sigmoid or tanh neurons.

I've painted a picture of uncertainty here, stressing that we do not yet have a solid theory of how activation functions should be chosen. Indeed, the problem is harder even than I have described, for there are infinitely many possible activation functions. Which is the best for any given problem? Which will result in a network which learns fastest? Which will give the highest test accuracies? I am surprised how little really deep and systematic investigation has been done of these questions. Ideally, we'd have a theory which tells us, in detail, how to choose (and perhaps modify-on-the-fly) our activation functions. On the other hand, we shouldn't let the lack of a full theory stop us! We have powerful tools already at hand, and can make a lot of progress with those tools. Through the remainder of this book I'll continue to use sigmoid neurons as our go-to neuron, since they're powerful and provide concrete illustrations of the core ideas about neural nets. But keep in the back of your mind that these same ideas can be applied to other types of neuron, and that there are sometimes advantages in doing so.

On stories in neural networks

Question: *How do you approach utilizing and researching machine learning techniques that are supported almost entirely empirically, as opposed to mathematically? Also in what situations have you noticed some of these techniques fail?*

> **Answer:** You have to realize that our theoretical tools are very weak. Sometimes, we have good mathematical intuitions for why a particular technique should work. Sometimes our intuition ends up being wrong [...] The questions become: how well does my method work on this particular problem, and how large is the set of problems on which it works well.

— *Question and answer with neural networks researcher Yann LeCun*

Once, attending a conference on the foundations of quantum mechanics, I noticed what seemed to me a most curious verbal habit: when talks finished, questions from the audience often began with "I'm very sympathetic to your point of view, but [...]". Quantum foundations was not my usual field, and I noticed this style of questioning because at other scientific conferences I'd rarely or never heard a questioner express their sympathy for the point of view of the speaker. At the time, I thought the prevalence of the question suggested that little genuine progress was being made in quantum foundations, and people were merely spinning their wheels. Later, I realized that assessment was too harsh. The speakers were wrestling with some of the hardest problems human minds have ever confronted. Of course progress was slow! But there was still value in hearing updates on how people were thinking, even if they didn't always have unarguable new progress to report.

Boltzmann Machines, by Vinod Nair and Geoffrey Hinton (2010), which demonstrates the benefits of using rectified linear units in a somewhat different approach to neural networks.

You may have noticed a verbal tic similar to “I’m very sympathetic [...]” in the current book. To explain what we’re seeing I’ve often fallen back on saying “Heuristically, [...]”, or “Roughly speaking, [...]”, following up with a story to explain some phenomenon or other. These stories are plausible, but the empirical evidence I’ve presented has often been pretty thin. If you look through the research literature you’ll see that stories in a similar style appear in many research papers on neural nets, often with thin supporting evidence. What should we think about such stories?

In many parts of science - especially those parts that deal with simple phenomena – it’s possible to obtain very solid, very reliable evidence for quite general hypotheses. But in neural networks there are large numbers of parameters and hyper-parameters, and extremely complex interactions between them. In such extraordinarily complex systems it’s exceedingly difficult to establish reliable general statements. Understanding neural networks in their full generality is a problem that, like quantum foundations, tests the limits of the human mind. Instead, we often make do with evidence for or against a few specific instances of a general statement. As a result those statements sometimes later need to be modified or abandoned, when new evidence comes to light.

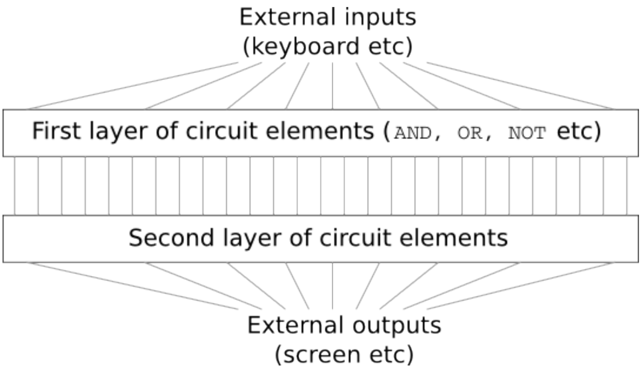
One way of viewing this situation is that any heuristic story about neural networks carries with it an implied challenge. For example, consider the statement I quoted earlier, explaining why dropout works[41]: “This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.” This is a rich, provocative statement, and one could build a fruitful research program entirely around unpacking the statement, figuring out what in it is true, what is false, what needs variation and refinement. Indeed, there is now a small industry of researchers who are investigating dropout (and many variations), trying to understand how it works, and what its limits are. And so it goes with many of the heuristics we’ve discussed. Each heuristic is not just a (potential) explanation, it’s also a challenge to investigate and understand in more detail.

Of course, there is not time for any single person to investigate all these heuristic explanations in depth. It’s going to take decades (or longer) for the community of neural networks researchers to develop a really powerful, evidence-based theory of how neural networks learn. Does this mean you should reject heuristic explanations as unrigorous, and not sufficiently evidence-based? No! In fact, we need such heuristics to inspire and guide our thinking. It’s like the great age of exploration: the early explorers sometimes explored (and made new discoveries) on the basis of beliefs which were wrong in important ways. Later, those mistakes were corrected as we filled in our knowledge of geography. When you understand something poorly - as the explorers understood geography, and as we understand neural nets today – it’s more important to explore boldly than it is to be rigorously correct in every step of your thinking. And so you should view these stories as a useful guide to how to think about neural nets, while retaining a healthy awareness of the limitations of such stories, and carefully keeping track of just how strong the evidence is for any given line of reasoning. Put another way, we need good stories to help motivate and inspire us, and rigorous in-depth investigation in order to uncover the real facts of the matter.

[41] [link]From ImageNet Classification with Deep Convolutional Neural Networks by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton (2012).

4 Why are deep neural networks hard to train?

Imagine you're an engineer who has been asked to design a computer from scratch. One day you're working away in your office, designing logical circuits, setting out AND gates, OR gates, and so on, when your boss walks in with bad news. The customer has just added a surprising design requirement: the circuit for the entire computer must be just two layers deep:


External inputs
(keyboard etc)
First layer of circuit elements (AND, OR, NOT etc)
Second layer of circuit elements
External outputs
(screen etc)

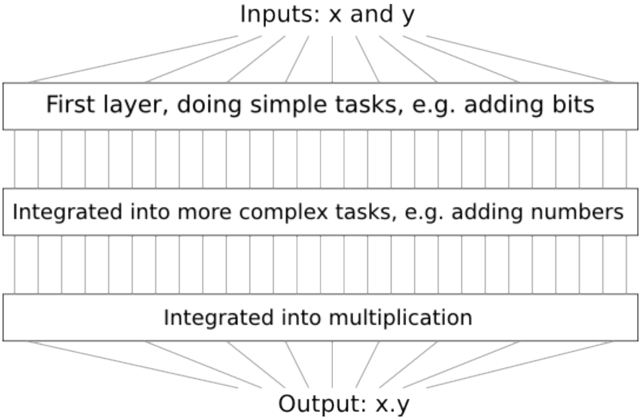
You're dumbfounded, and tell your boss: "The customer is crazy!"

Your boss replies: "I think they're crazy, too. But what the customer wants, they get."

In fact, there's a limited sense in which the customer isn't crazy. Suppose you're allowed to use a special logical gate which lets you AND together as many inputs as you want. And you're also allowed a many-input NAND gate, that is, a gate which can AND multiple inputs and then negate the output. With these special gates it turns out to be possible to compute any function at all using a circuit that's just two layers deep.

But just because something is possible doesn't make it a good idea. In practice, when solving circuit design problems (or most any kind of algorithmic problem), we usually start by figuring out how to solve sub-problems, and then gradually integrate the solutions. In other words, we build up to a solution through multiple layers of abstraction.

For instance, suppose we're designing a logical circuit to multiply two numbers. Chances are we want to build it up out of sub-circuits doing operations like adding two numbers. The sub-circuits for adding two numbers will, in turn, be built up out of sub-sub-circuits for adding two bits. Very roughly speaking our circuit will look like:

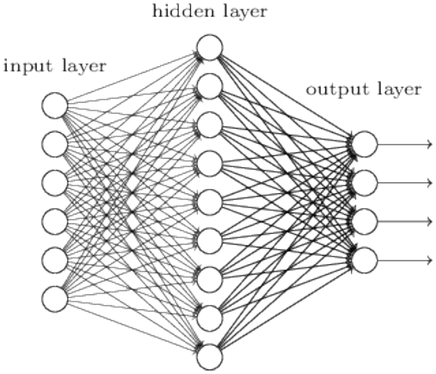

Inputs: x and y
First layer, doing simple tasks, e.g. adding bits
Integrated into more complex tasks, e.g. adding numbers
Integrated into multiplication
Output: x.y

That is, our final circuit contains at least three layers of circuit elements. In fact, it'll probably contain more than three layers, as we break the sub-tasks down into smaller units than I've described. But you get the general idea.

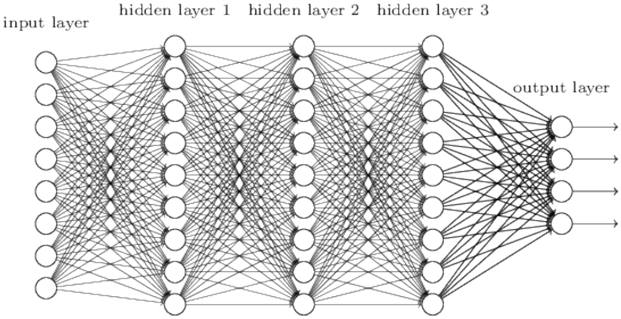
So deep circuits make the process of design easier. But they're not just helpful for design. There are, in fact, mathematical proofs showing that for some functions very shallow circuits require exponentially more circuit elements to compute than do deep circuits. For instance, a famous series of papers in the early 1980s[1] showed that computing the parity of a set of bits requires exponentially many gates, if done with a shallow circuit. On the other hand, if you use deeper circuits it's easy to compute the parity using a small circuit: you just compute the parity of pairs of bits, then use those results to compute the parity of pairs of pairs of bits, and so on, building up quickly to the overall parity. Deep circuits thus can be intrinsically much more powerful than shallow circuits.

Up to now, this book has approached neural networks like the crazy customer. Almost all the networks we've worked with have just a single hidden layer of neurons (plus the input and output layers):

[1] The history is somewhat complex, so I won't give detailed references. See Johan Håstad's 2012 paper [link]On the correlation of parity and small-depth circuits for an account of the early history and references.



These simple networks have been remarkably useful: in earlier chapters we used networks like this to classify handwritten digits with better than 98 percent accuracy! Nonetheless, intuitively we'd expect networks with many more hidden layers to be more powerful:



Such networks could use the intermediate layers to build up multiple layers of abstraction, just as we do in Boolean circuits. For instance, if we're doing visual pattern recognition, then the neurons in the first layer might learn to recognize edges, the neurons in the second layer could learn to recognize more complex shapes, say triangle or rectangles, built up from edges. The third layer would then recognize still more complex shapes. And so on. These multiple layers of abstraction seem likely to give deep networks a compelling advantage in learning to solve complex pattern recognition problems. Moreover, just as in the case of circuits, there are theoretical results suggesting that deep networks are intrinsically more powerful than shallow networks[2].

How can we train such deep networks? In this chapter, we'll try training deep networks using our workhorse learning algorithm - stochastic gradient descent by backpropagation. But we'll run into trouble, with our deep networks not performing much (if at all) better than shallow networks.

That failure seems surprising in the light of the discussion above. Rather than give up on deep networks, we'll dig down and try to understand what's making our deep networks hard to train. When we look closely, we'll discover that the different layers in our deep network are learning at vastly different speeds. In particular, when later layers in the network are learning well, early layers often get stuck during training, learning almost nothing at all. This

[2] For certain problems and network architectures this is proved in [link]On the number of response regions of deep feed forward networks with piece-wise linear activations, by Razvan Pascanu, Guido Montúfar, and Yoshua Bengio (2014). See also the more informal discussion in section 2 of [link]Learning deep architectures for AI, by Yoshua Bengio (2009).

stuckness isn't simply due to bad luck. Rather, we'll discover there are fundamental reasons the learning slowdown occurs, connected to our use of gradient-based learning techniques.

As we delve into the problem more deeply, we'll learn that the opposite phenomenon can also occur: the early layers may be learning well, but later layers can become stuck. In fact, we'll find that there's an intrinsic instability associated to learning by gradient descent in deep, many-layer neural networks. This instability tends to result in either the early or the later layers getting stuck during training.

This all sounds like bad news. But by delving into these difficulties, we can begin to gain insight into what's required to train deep networks effectively. And so these investigations are good preparation for the next chapter, where we'll use deep learning to attack image recognition problems.

4.1 The vanishing gradient problem

So, what goes wrong when we try to train a deep network?

To answer that question, let's first revisit the case of a network with just a single hidden layer. As per usual, we'll use the MNIST digit classification problem as our playground for learning and experimentation[3].

If you wish, you can follow along by training networks on your computer. It is also, of course, fine to just read along. If you do wish to follow live, then you'll need Python 2.7, Numpy, and a copy of the code, which you can get by cloning the relevant repository from the command line:

```
git clone https://github.com/mnielsen/neural-networks-and-deep-learning.git
```

If you don't use git then you can download the data and code here. You'll need to change into the `src` subdirectory. Then, from a Python shell we load the MNIST data:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
```

We set up our network:

```
>>> import network2
>>> net = network2.Network([784, 30, 10])
```

This network has 784 neurons in the input layer, corresponding to the $28 \times 28 = 784$ pixels in the input image. We use 30 hidden neurons, as well as 10 output neurons, corresponding to the 10 possible classifications for the MNIST digits ('0', '1', '2',..., '9').

Let's try training our network for 30 complete epochs, using mini-batches of 10 training examples at a time, a learning rate Îů=0.1, and regularization parameter $\lambda = 5.0$. As we train we'll monitor the classification accuracy on the `validation_data`[4]:

```
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

[3] I introduced the MNIST problem and data here and here.

[4] Note that the networks is likely to take some minutes to train, depending on the speed of your machine. So if you're running the code you may wish to continue reading and return later, not wait for the code to finish executing.

We get a classification accuracy of 96.48 percent (or thereabouts – it'll vary a bit from run to run), comparable to our earlier results with a similar configuration.

Now, let's add another hidden layer, also with 30 neurons in it, and try training with the same hyper-parameters:

```
>>> net = network2.Network([784, 30, 30, 10])
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

This gives an improved classification accuracy, 96.90 percent. That's encouraging: a little more depth is helping. Let's add another 30-neuron hidden layer:

```
>>> net = network2.Network([784, 30, 30, 30, 10])
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

That doesn't help at all. In fact, the result drops back down to 96.57 percent, close to our original shallow network. And suppose we insert one further hidden layer:

```
>>> net = network2.Network([784, 30, 30, 30, 30, 10])
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

The classification accuracy drops again, to 96.53 percent. That's probably not a statistically significant drop, but it's not encouraging, either.

This behaviour seems strange. Intuitively, extra hidden layers ought to make the network able to learn more complex classification functions, and thus do a better job classifying. Certainly, things shouldn't get worse, since the extra layers can, in the worst case, simply do nothing[5]. But that's not what's going on.

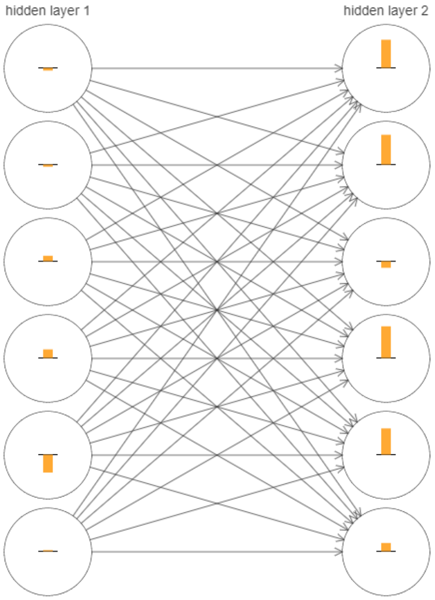
So what is going on? Let's assume that the extra hidden layers really could help in principle, and the problem is that our learning algorithm isn't finding the right weights and biases. We'd like to figure out what's going wrong in our learning algorithm, and how to do better.

To get some insight into what's going wrong, let's visualize how the network learns. Below, I've plotted part of a [784,30,30,10] network, i.e., a network with two hidden layers, each containing 30 hidden neurons. Each neuron in the diagram has a little bar on it, representing how quickly that neuron is changing as the network learns. A big bar means the neuron's weights and bias are changing rapidly, while a small bar means the weights and bias are changing slowly. More precisely, the bars denote the gradient $\partial C/\partial b$ for each neuron, i.e., the rate of change of the cost with respect to the neuron's bias. Back in Chapter 2 we saw that this gradient quantity controlled not just how rapidly the bias changes during learning, but also how rapidly the weights input to the neuron change, too. Don't worry if you don't recall the details: the thing to keep in mind is simply that these bars show how quickly each neuron's weights and bias are changing as the network learns.

To keep the diagram simple, I've shown just the top six neurons in the two hidden layers. I've omitted the input neurons, since they've got no weights or biases to learn. I've also omitted the output neurons, since we're doing layer-wise comparisons, and it makes most sense to compare layers with the same number of neurons. The results are plotted at the very beginning of training, i.e., immediately after the network is initialized. Here they are[6]:

[5] See this later problem to understand how to build a hidden layer that does nothing.

[6] The data plotted is generated using the program `generate_gradient.py`. The same program is also used to generate the results quoted later in this section.



The network was initialized randomly, and so it's not surprising that there's a lot of variation in how rapidly the neurons learn. Still, one thing that jumps out is that the bars in the second hidden layer are mostly much larger than the bars in the first hidden layer. As a result, the neurons in the second hidden layer will learn quite a bit faster than the neurons in the first hidden layer. Is this merely a coincidence, or are the neurons in the second hidden layer likely to learn faster than neurons in the first hidden layer in general?

To determine whether this is the case, it helps to have a global way of comparing the speed of learning in the first and second hidden layers. To do this, let's denote the gradient as $\delta^l_j = \partial C/\partial b^l_j$, i.e., the gradient for the j-th neuron in the l-th layer[7] We can think of the gradient δ^1 as a vector whose entries determine how quickly the first hidden layer learns, and δ^2 as a vector whose entries determine how quickly the second hidden layer learns. We'll then use the lengths of these vectors as (rough!) global measures of the speed at which the layers are learning. So, for instance, the length $\|\delta^1\|$ measures the speed at which the first hidden layer is learning, while the length $\|\delta^2\|$ measures the speed at which the second hidden layer is learning.

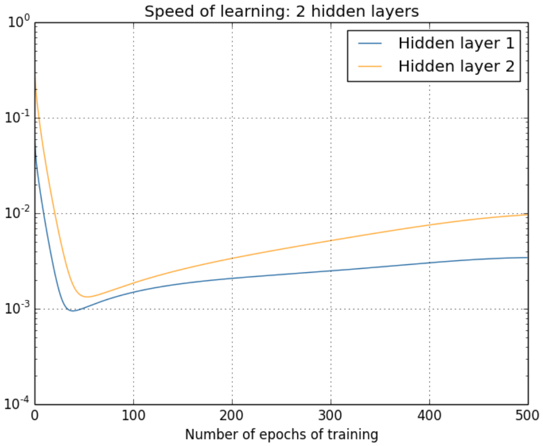
With these definitions, and in the same configuration as was plotted above, we find $\|\delta^1\| = 0.07\ldots$ and $\|\delta^2\| = 0.31\ldots$. So this confirms our earlier suspicion: the neurons in the second hidden layer really are learning much faster than the neurons in the first hidden layer.

What happens if we add more hidden layers? If we have three hidden layers, in a [784,30,30,30,10] network, then the respective speeds of learning turn out to be 0.012, 0.060, and 0.283. Again, earlier hidden layers are learning much slower than later hidden layers. Suppose we add yet another layer with 30 hidden neurons. In that case, the respective speeds of learning are 0.003, 0.017, 0.070, and 0.285. The pattern holds: early layers learn slower than later layers.

We've been looking at the speed of learning at the start of training, that is, just after the networks are initialized. How does the speed of learning change as we train our networks?

[7] Back in Chapter 2 we referred to this as the error, but here we'll adopt the informal term "gradient". I say "informal" because of course this doesn't explicitly include the partial derivatives of the cost with respect to the weights, $\partial C/\partial w$.

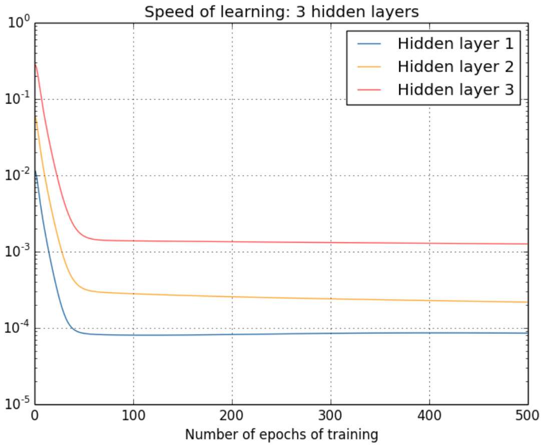
Let's return to look at the network with just two hidden layers. The speed of learning changes as follows:



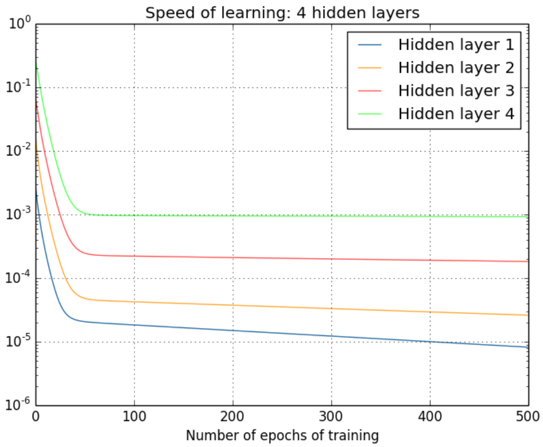
To generate these results, I used batch gradient descent with just 1,000 training images, trained over 500 epochs. This is a bit different than the way we usually train - I've used no mini-batches, and just 1,000 training images, rather than the full 50,000 image training set. I'm not trying to do anything sneaky, or pull the wool over your eyes, but it turns out that using mini-batch stochastic gradient descent gives much noisier (albeit very similar, when you average away the noise) results. Using the parameters I've chosen is an easy way of smoothing the results out, so we can see what's going on.

In any case, as you can see the two layers start out learning at very different speeds (as we already know). The speed in both layers then drops very quickly, before rebounding. But through it all, the first hidden layer learns much more slowly than the second hidden layer.

What about more complex networks? Here's the results of a similar experiment, but this time with three hidden layers (a [784,30,30,30,10] network):



Again, early hidden layers learn much more slowly than later hidden layers. Finally, let's add a fourth hidden layer (a [784,30,30,30,30,10] network), and see what happens when we train:



Again, early hidden layers learn much more slowly than later hidden layers. In this case, the first hidden layer is learning roughly 100 times slower than the final hidden layer. No wonder we were having trouble training these networks earlier!

We have here an important observation: in at least some deep neural networks, the gradient tends to get smaller as we move backward through the hidden layers. This means that neurons in the earlier layers learn much more slowly than neurons in later layers. And while we've seen this in just a single network, there are fundamental reasons why this happens in many neural networks. The phenomenon is known as the vanishing gradient problem[8].

Why does the vanishing gradient problem occur? Are there ways we can avoid it? And how should we deal with it in training deep neural networks? In fact, we'll learn shortly that it's not inevitable, although the alternative is not very attractive, either: sometimes the gradient gets much larger in earlier layers! This is the exploding gradient problem, and it's not much better news than the vanishing gradient problem. More generally, it turns out that the gradient in deep neural networks is unstable, tending to either explode or vanish in earlier layers. This instability is a fundamental problem for gradient-based learning in deep neural networks. It's something we need to understand, and, if possible, take steps to address.

One response to vanishing (or unstable) gradients is to wonder if they're really such a problem. Momentarily stepping away from neural nets, imagine we were trying to numerically minimize a function $f(x)$ of a single variable. Wouldn't it be good news if the derivative $f'(x)$ was small? Wouldn't that mean we were already near an extremum? In a similar way, might the small gradient in early layers of a deep network mean that we don't need to do much adjustment of the weights and biases?

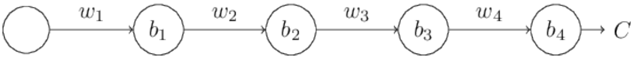
Of course, this isn't the case. Recall that we randomly initialized the weight and biases in the network. It is extremely unlikely our initial weights and biases will do a good job at

[8] See [link]Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, by Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber (2001). This paper studied recurrent neural nets, but the essential phenomenon is the same as in the feedforward networks we are studying. See also [link]Sepp Hochreiter's earlier Diploma Thesis, Untersuchungen zu dynamischen neuronalen Netzen (1991, in German).

whatever it is we want our network to do. To be concrete, consider the first layer of weights in a [784,30,30,30,10] network for the MNIST problem. The random initialization means the first layer throws away most information about the input image. Even if later layers have been extensively trained, they will still find it extremely difficult to identify the input image, simply because they don't have enough information. And so it can't possibly be the case that not much learning needs to be done in the first layer. If we're going to train deep networks, we need to figure out how to address the vanishing gradient problem.

4.2 What's causing the vanishing gradient problem? Unstable gradients in deep neural nets

To get insight into why the vanishing gradient problem occurs, let's consider the simplest deep neural network: one with just a single neuron in each layer. Here's a network with three hidden layers:

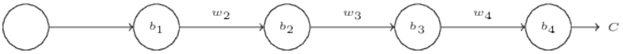


Here, $w_1, w_2, \ldots$ are the weights, $b_1, b_2, \ldots$ are the biases, and C is some cost function. Just to remind you how this works, the output a_j from the j-th neuron is $\sigma(z_j)$, where σ is the usual sigmoid activation function, and $z_j = w_j a_{j1} + b_j$ is the weighted input to the neuron. I've drawn the cost C at the end to emphasize that the cost is a function of the network's output, a_4: if the actual output from the network is close to the desired output, then the cost will be low, while if it's far away, the cost will be high.

We're going to study the gradient $\partial C/\partial b_1$ associated to the first hidden neuron. We'll figure out an expression for $\partial C/\partial b_1$, and by studying that expression we'll understand why the vanishing gradient problem occurs.

I'll start by simply showing you the expression for $\partial C/\partial b_1$. It looks forbidding, but it's actually got a simple structure, which I'll describe in a moment. Here's the expression (ignore the network, for now, and note that σ' is just the derivative of the σ function):

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



The structure in the expression is as follows: there is a $\sigma'(z_j)$ term in the product for each neuron in the network; a weight w_j term for each weight in the network; and a final $\partial C/\partial a_4$ term, corresponding to the cost function at the end. Notice that I've placed each term in the expression above the corresponding part of the network. So the network itself is a mnemonic for the expression.

You're welcome to take this expression for granted, and skip to the discussion of how it relates to the vanishing gradient problem. There's no harm in doing this, since the expression is a special case of our earlier discussion of backpropagation. But there's also a simple explanation of why the expression is true, and so it's fun (and perhaps enlightening) to take a look at that explanation.

Imagine we make a small change δb_1 in the bias b_1. That will set off a cascading series of changes in the rest of the network. First, it causes a change δa_1 in the output from the first hidden neuron. That, in turn, will cause a change δz_2 in the weighted input to the second hidden neuron. Then a change δa_2 in the output from the second hidden neuron. And so on, all the way through to a change δC in the cost at the output. We have

$$\frac{\partial C}{\partial b_1} \approx \frac{\Delta C}{\Delta b_1}. \tag{4.1}$$

This suggests that we can figure out an expression for the gradient $\partial C/\partial b_1$ by carefully tracking the effect of each step in this cascade.

To do this, let's think about how δb_1 causes the output a1 from the first hidden neuron to change. We have $a_1 = \sigma(z_1) = \sigma(w_1 a_0 + b1)$, so

$$\Delta a_1 \approx \frac{\partial \sigma(w_1 a_0 + b_1)}{\partial b_1} \Delta b_1 = \sigma'(z_1) \Delta b_1. \tag{4.2}$$

That $\sigma'(z_1)$ term should look familiar: it's the first term in our claimed expression for the gradient $\partial C/\partial b_1$. Intuitively, this term converts a change δb_1 in the bias into a change δa_1 in the output activation. That change δa_1 in turn causes a change in the weighted input $z_2 = w_2 a_1 + b_2$ to the second hidden neuron:

$$\Delta z_2 \quad \approx \tfrac{\partial z_2}{\partial a_1} \Delta a_1 = w_2 \Delta a_1. \tag{4.3}$$

Combining our expressions for δz_2 and δa_1, we see how the change in the bias b_1 propagates along the network to affect z_2:

$$\Delta z_2 \approx \sigma'(z_1) w_2 \Delta b_1. \tag{4.4}$$

Again, that should look familiar: we've now got the first two terms in our claimed expression for the gradient $\partial C/\partial b_1$.

We can keep going in this fashion, tracking the way changes propagate through the rest of the network. At each neuron we pick up a $\sigma'(z_j)$ term, and through each weight we pick up a w_j term. The end result is an expression relating the final change δC in cost to the initial change δb_1 in the bias:

$$\Delta C \approx \sigma'(z_1) w_2 \sigma'(z_2) \ldots \sigma'(z_4) \frac{\partial C}{\partial a_4} \Delta b_1. \tag{4.5}$$

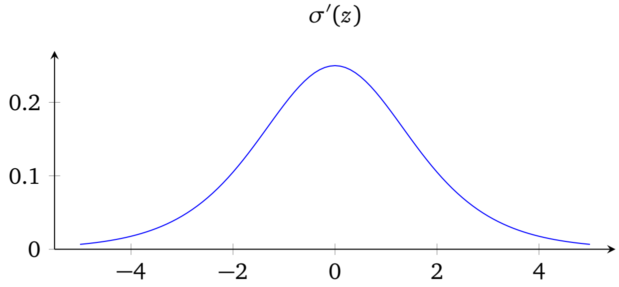
Dividing by δb_1 we do indeed get the desired expression for the gradient:

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) w_2 \sigma'(z_2) \ldots \sigma'(z_4) \frac{\partial C}{\partial a_4}. \tag{4.6}$$

Why the vanishing gradient problem occurs: To understand why the vanishing gradient problem occurs, let's explicitly write out the entire expression for the gradient:

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\, w_2 \sigma'(z_2)\, w_3 \sigma'(z_3)\, w_4 \sigma'(z_4)\, \frac{\partial C}{\partial a_4}. \tag{4.7}$$

Excepting the very last term, this expression is a product of terms of the form $w_j \sigma'(z_j)$. To understand how each of those terms behave, let's look at a plot of the function σ':



The derivative reaches a maximum at $\sigma'(0) = 1/4$. Now, if we use our standard approach to initializing the weights in the network, then we'll choose the weights using a Gaussian with mean 0 and standard deviation 1. So the weights will usually satisfy $|w_j| < 1$. Putting these observations together, we see that the terms $w_j \sigma'(z_j)$ will usually satisfy $|w_j \sigma'(z_j)| < 1/4$. And when we take a product of many such terms, the product will tend to exponentially decrease: the more terms, the smaller the product will be. This is starting to smell like a possible explanation for the vanishing gradient problem.

To make this all a bit more explicit, let's compare the expression for $\partial C/\partial b_1$ to an expression for the gradient with respect to a later bias, say $\partial C/\partial b_3$. Of course, we haven't explicitly worked out an expression for $\partial C/\partial b_3$, but it follows the same pattern described above for $\partial C/\partial b_1$. Here's the comparison of the two expressions:

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\, \overbrace{w_2 \sigma'(z_2)}^{< \frac{1}{4}}\, \overbrace{w_3 \sigma'(z_3)}^{< \frac{1}{4}}\, \underbrace{w_4 \sigma'(z_4) \frac{\partial C}{\partial a_4}}_{\text{common terms}}$$

$$\frac{\partial C}{\partial b_3} = \sigma'(z_3)\, \overbrace{w_4 \sigma'(z_4) \frac{\partial C}{\partial a_4}}^{\text{common terms}}$$

The two expressions share many terms. But the gradient $\partial C/\partial b_1$ includes two extra terms each of the form $w_j \sigma'(z_j)$. As we've seen, such terms are typically less than $1/4$ in magnitude. And so the gradient $\partial C/\partial b_1$ will usually be a factor of 16 (or more) smaller than $\partial C/\partial b_3$. This is the essential origin of the vanishing gradient problem.

Of course, this is an informal argument, not a rigorous proof that the vanishing gradient problem will occur. There are several possible escape clauses. In particular, we might wonder whether the weights w_j could grow during training. If they do, it's possible the terms $w_j \sigma'(z_j)$ in the product will no longer satisfy $|w_j \sigma'(z_j)| < 1/4$. Indeed, if the terms get large enough – greater than 1 – then we will no longer have a vanishing gradient problem. Instead, the gradient will actually grow exponentially as we move backward through the layers. Instead of a vanishing gradient problem, we'll have an exploding gradient problem.

The exploding gradient problem: Let's look at an explicit example where exploding gradients occur. The example is somewhat contrived: I'm going to fix parameters in the network in just the right way to ensure we get an exploding gradient. But even though the example is contrived, it has the virtue of firmly establishing that exploding gradients aren't merely a hypothetical possibility, they really can happen.

There are two steps to getting an exploding gradient. First, we choose all the weights in the network to be large, say $w_1 = w_2 = w_3 = w_4 = 100$. Second, we'll choose the biases so that the $\sigma'(z_j)$ terms are not too small. That's actually pretty easy to do: all we need do is choose the biases to ensure that the weighted input to each neuron is $z_j = 0$ (and so $\sigma'(z_j) = 1/4$). So, for instance, we want $z_1 = w_1 a_0 + b1 = 0$. We can achieve this by setting $b_1 = 100 * a0$. We can use the same idea to select the other biases. When we do this, we see that all the terms $w_j \sigma'(z_j)$ are equal to $100 * 1/4 = 25$. With these choices we get an exploding gradient.

The unstable gradient problem: The fundamental problem here isn't so much the vanishing gradient problem or the exploding gradient problem. It's that the gradient in early layers is the product of terms from all the later layers. When there are many layers, that's an intrinsically unstable situation. The only way all layers can learn at close to the same speed is if all those products of terms come close to balancing out. Without some mechanism or underlying reason for that balancing to occur, it's highly unlikely to happen simply by chance. In short, the real problem here is that neural networks suffer from an unstable gradient problem. As a result, if we use standard gradient-based learning techniques, different layers in the network will tend to learn at wildly different speeds.

Exercise

- In our discussion of the vanishing gradient problem, we made use of the fact that $|\sigma'(z)| < 1/4$. Suppose we used a different activation function, one whose derivative could be much larger. Would that help us avoid the unstable gradient problem?

The prevalence of the vanishing gradient problem: We've seen that the gradient can either vanish or explode in the early layers of a deep network. In fact, when using sigmoid neurons the gradient will usually vanish. To see why, consider again the expression $|w\sigma'(z)|$. To avoid the vanishing gradient problem we need $|w\sigma'(z)| \geq 1$. You might think this could happen easily if w is very large. However, it's more difficult than it looks. The reason is that the $\sigma'(z)$ term also depends on w: $\sigma'(z) = \sigma'(wa + b)$, where a is the input activation. So when we make w large, we need to be careful that we're not simultaneously making $\sigma'(wa + b)$ small. That turns out to be a considerable constraint. The reason is that when we make w large we tend to make wa+b very large. Looking at the graph of σ' you can see that this puts us off in the "wings" of the σ' function, where it takes very small values. The only way to avoid this is if the input activation falls within a fairly narrow range of values (this qualitative explanation is made quantitative in the first problem below). Sometimes that will chance to happen. More often, though, it does not happen. And so in the generic case we have vanishing gradients.