Neural Networks and Deep Learning

Michael Nielsen

August 6, 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.

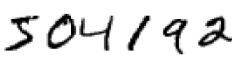
This hands-on approach means that you'll need some programming experience to read the book. But you don't need to be a professional programmer. I've written the code in Python (version 2.7), which, even if you don't program in Python, should be easy to understand with just a little effort. Through the 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.

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.

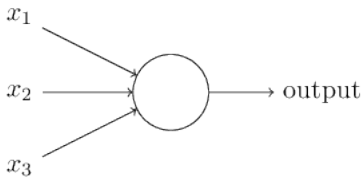
Of course, if the point of the chapter was only to write a computer program to recognize handwritten digits, then the chapter would be much shorter! But along the way we'll develop many key ideas about neural networks, including two important types of artificial neuron (the perceptron and the sigmoid 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:

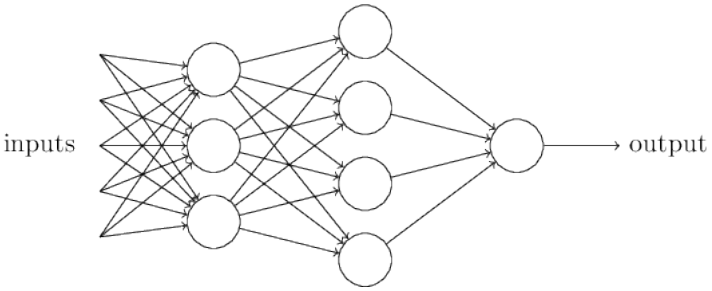
1. Is the weather good?
2. Does your boyfriend or girlfriend want to accompany you?
3. Is the festival near public transit? (You don't own a car).

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.

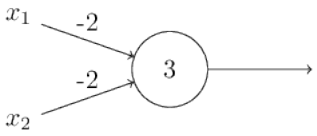
Let's simplify the way we describe perceptrons. The condition $\sum_j w_j x_j >$threshold is cumbersome, and we can make two notational changes to simplify it. The first change is to write $\sum_j w_j x_j$ as a dot product, $w \cdot x = \sum_j w_j x_j$, where w and x are vectors whose components are 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

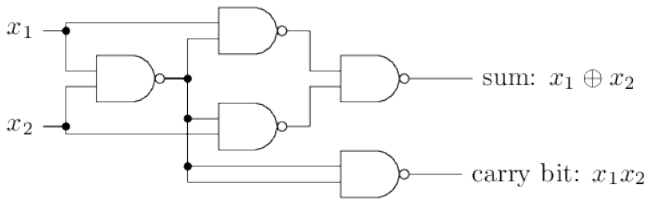
perceptrons, but we'll see later that it leads to further notational simplifications. Because of this, in the remainder of the book we won't use the threshold, we'll always use the bias.

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:

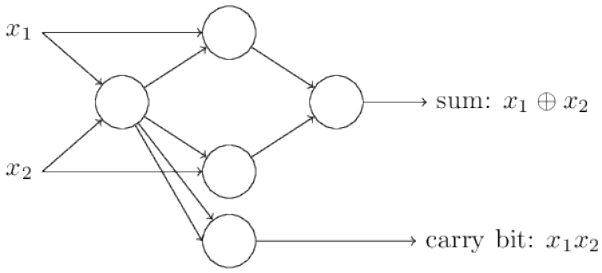


Then we see that input 00 produces output 1, since $(-2)*0+(-2)*0+3=3$ is positive. Here, I've introduced the $*$ symbol to make the multiplications explicit. Similar calculations show that the inputs 01 and 10 produce output 1. But the input 11 produces output 0, since $(-2)*1+(-2)*1+3=-1$ is negative. 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 $x_1 x_2$:

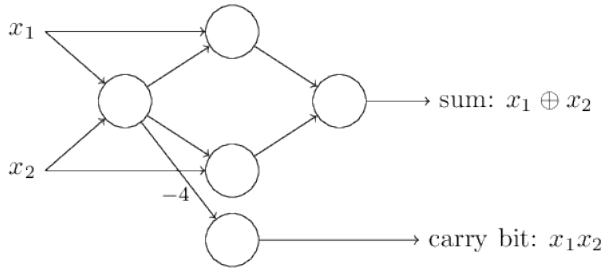


To get an equivalent network of perceptrons we replace all the NAND gates by perceptrons with two inputs, each with weight –2, and an overall bias of 3. Here's the resulting network. Note that I've moved the perceptron corresponding to the bottom right NAND gate a little, just to make it easier to draw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

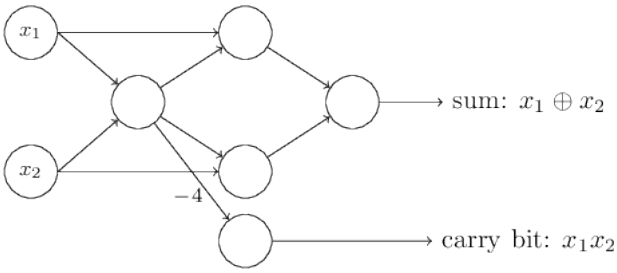


One notable aspect of this network of perceptrons is that the output from the leftmost perceptron is used twice as input to the bottommost perceptron. When I defined the perceptron

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.

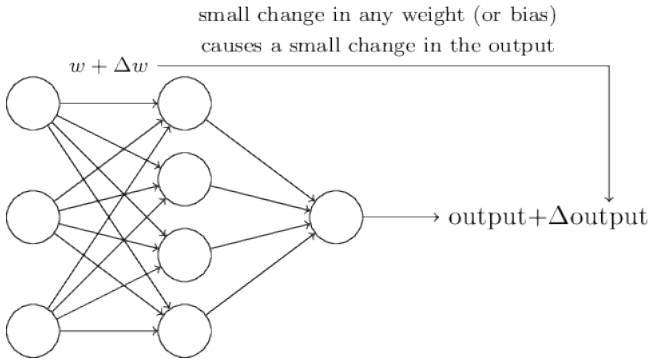
The computational universality of perceptrons is simultaneously reassuring and disappointing. It's reassuring because it tells us that networks of perceptrons can be as powerful as any other computing device. But it's also disappointing, because it makes it seem as though perceptrons are merely a new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



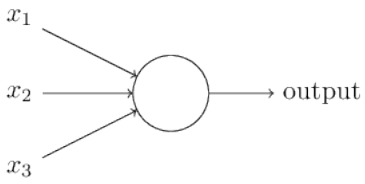
If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want. For example, suppose the network was mistakenly classifying an image as an "8" when it 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, $0.638lcdots$ 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,..., weights $w_1, w_2, \ldots$, and bias b is

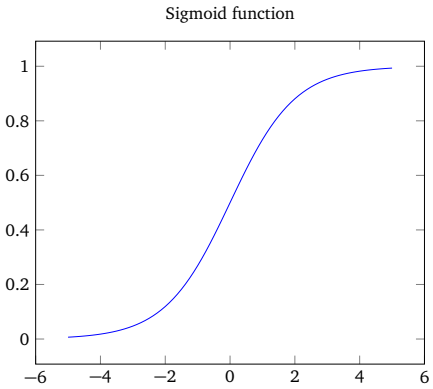
$$\frac{1}{1 + \exp\left(-\sum_j w_j x_j - b\right)}. \tag{1.4}$$

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.

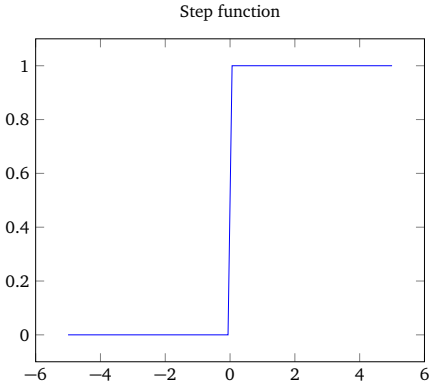
To understand the similarity to the perceptron model, suppose $z \equiv w \cdot x + b$ is a large positive number. Then $e^{-z} \approx 0$ and so $\sigma(z) \approx 1$. In other words, when $z = w \cdot x + b$ is large and positive, the output from the sigmoid neuron is approximately 1, just as it would have been for a perceptron. Suppose on the other hand that$z = w \cdot x + b$ is very negative. Then $e^{-z} \rightarrow \infty$, 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 ∂output$/\partial w_j$ and ∂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.

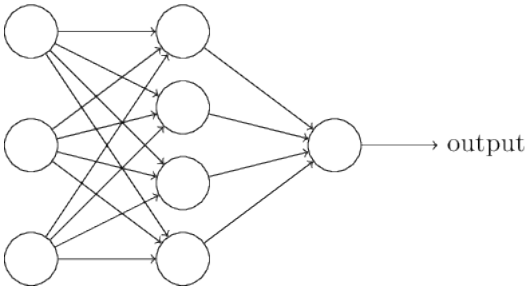
How should we interpret the output from a sigmoid neuron? Obviously, one big difference between perceptrons and sigmoid neurons is that sigmoid neurons don't just output 0 or 1. They can have as output any real number between 0 and 1, so values such as 0.173… and 0.689… are legitimate outputs. This can 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

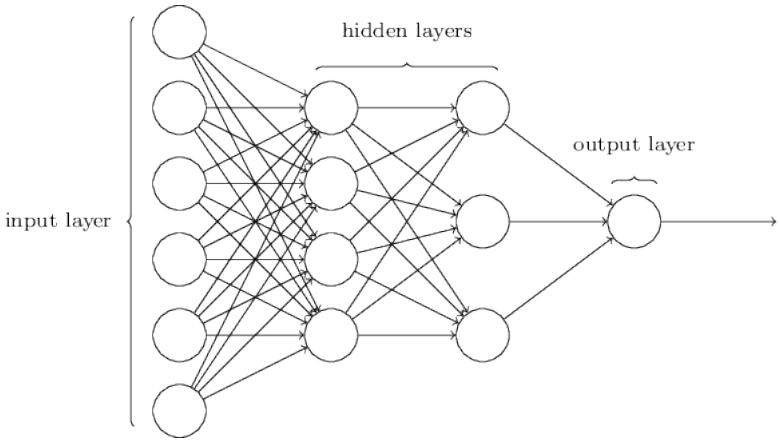
- **Sigmoid neurons simulating perceptrons, part I**
  Suppose we take all the weights and biases in a network of perceptrons, and multiply them by a positive constant, c>0. Show that the behavior of the network doesn't change.
- **Sigmoid neurons simulating perceptrons, part II**
  Suppose we have the 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.

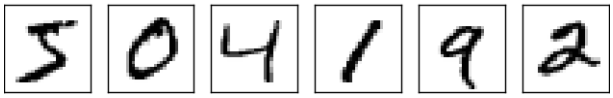
However, there are other models of artificial neural networks in which feedback loops are possible. These models are called recurrent neural networks. The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent. That firing can stimulate other neurons, 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,

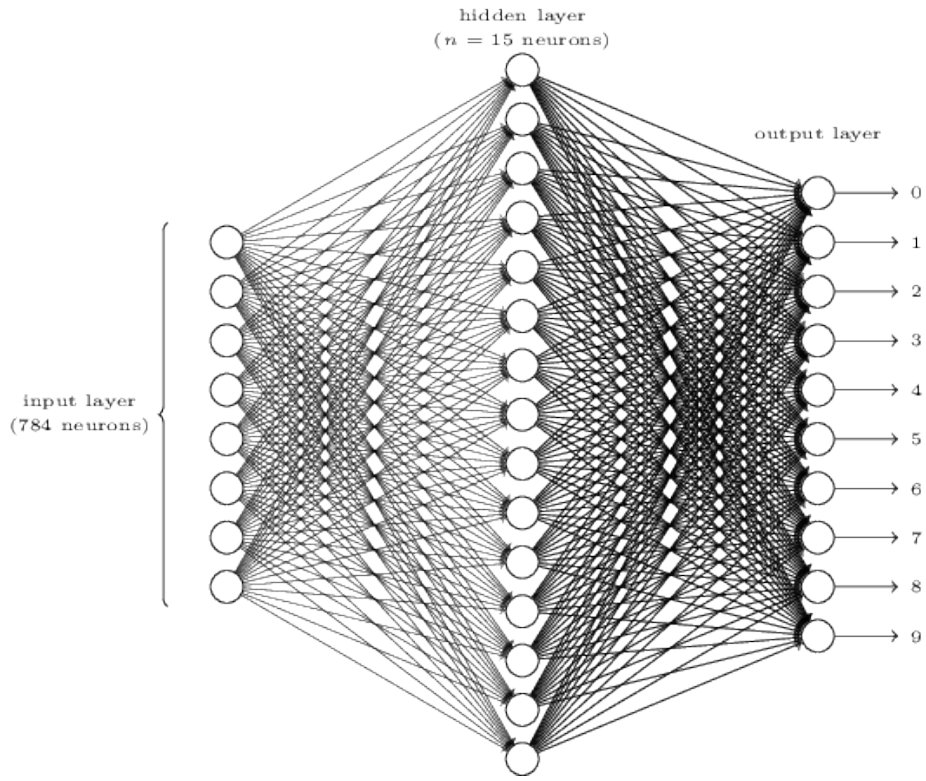


We humans solve this *segmentation problem* with ease, but it's challenging for a computer program to correctly break up the image. Once the image has been segmented, the program then needs to classify each individual digit. So, for instance, we'd like our program to recognize that the first digit above,

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

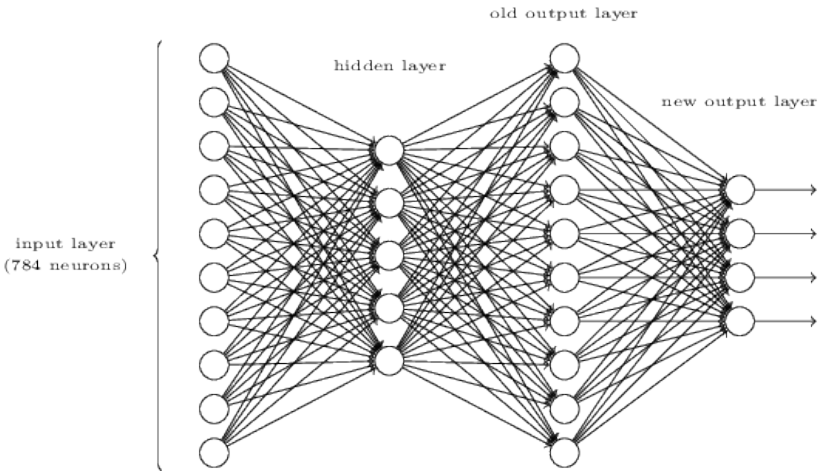
images, or slight distortions). But it seems safe to say that at least in this case we'd conclude that the input was a 0.

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

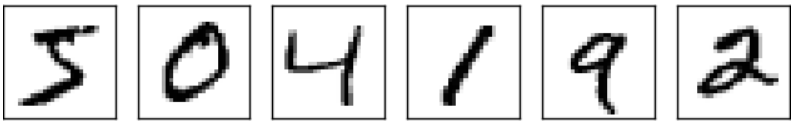
- There is a way of determining the bitwise representation of a digit by adding an extra layer to the three-layer network above. The extra layer converts the output from the previous layer into a binary representation, as illustrated in the figure below. Find a set of weights and biases for the new output 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

[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.

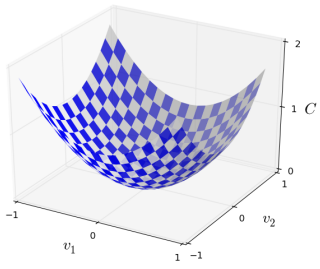
be to minimize the cost $C(w,b)$ as a function of the weights and biases. In other words, we want to find a set of weights and biases which make the cost as small as possible. We'll do that using an algorithm known as *gradient descent*.

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

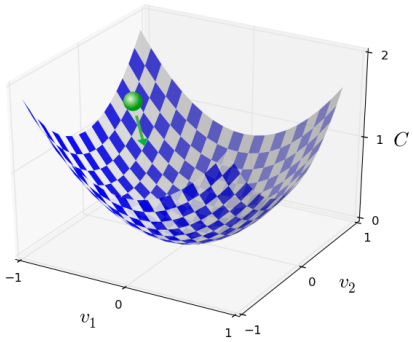
$$\Delta v = -\eta \nabla C, \tag{1.10}$$

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

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

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

By repeatedly applying this update rule we can "roll down the hill", and hopefully find a

[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.

minimum of the cost function. In other words, this is a rule which can be used to learn in a neural network.

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,

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

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

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.

Exercise

- 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 $w_k \to w'_k = w_k - \eta \partial C_x / \partial w_k$ and $b_l \to b'_l = b_l - \eta \partial C_x / \partial b_l$. 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

[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]).

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 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):
```

[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.

```
        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)
```

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

[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.

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
...
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?

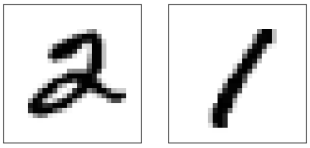
Earlier, I skipped over the details of how the MNIST data is loaded. It's pretty straightforward. For completeness, here's the code. The data structures used to store the MNIST data are described in the documentation strings – it's straightforward stuff, tuples and lists of Numpy ndarray objects (think 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:



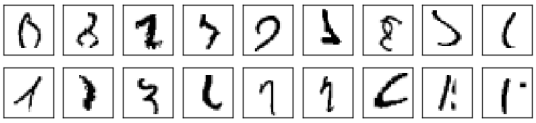
This suggests using the training data to compute average darknesses for each digit, 0,1,2,...,9. When presented with a new image, we compute how dark the image is, and then guess that it's whichever digit has the closest average darkness. This is a simple procedure, and is easy to code up, so I won't 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:

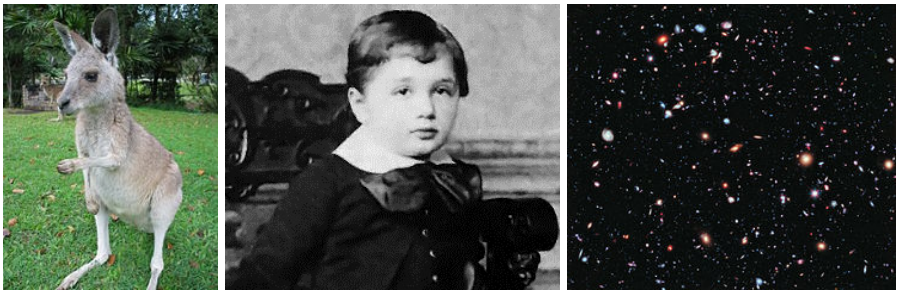


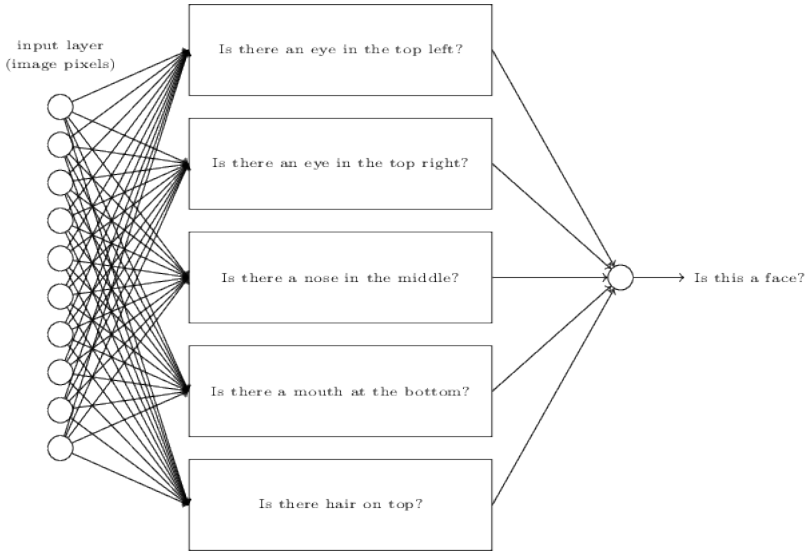
Figure 1.1: Credits: 1. Ester Inbar. 2. Unknown. 3. NASA, ESA, G. Illingworth, D. Magee, and P. Oesch (University of California, Santa Cruz), R. Bouwens (Leiden University), and the HUDF09 Team. [link]Click on the images for more details.es for more details.

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".

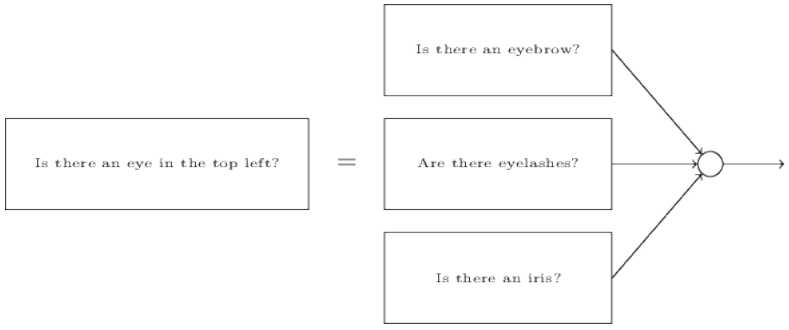
Let's suppose we do this, but that we're not using a learning algorithm. Instead, we're going to try to design a network by hand, choosing appropriate weights and biases. How might we go about it? Forgetting neural networks entirely for the moment, a heuristic we could use is to decompose the problem 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.

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:



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.

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?

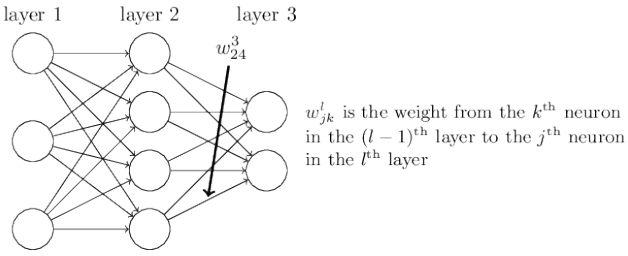
The reason, of course, is understanding. At the heart of backpropagation is an expression for the partial derivative $\partial C/\partial w$ of the cost function C with respect to any weight w (or bias b) in the network. The expression tells us how quickly the cost changes when we change the weights 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.

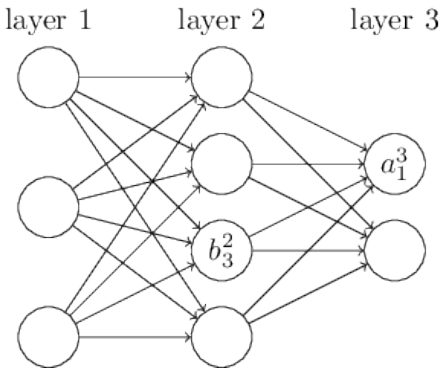
2.1 Warm up: a fast matrix-based approach to computing the output from a neural network

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:



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

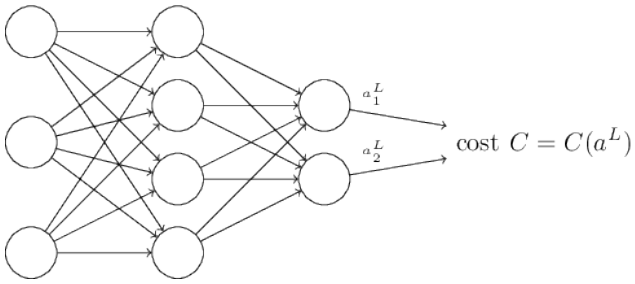
$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2, \tag{2.4}$$

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,

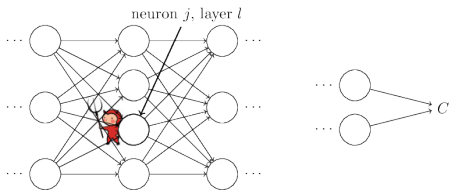
$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1*3 \\ 2*4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}. \tag{2.6}$$

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

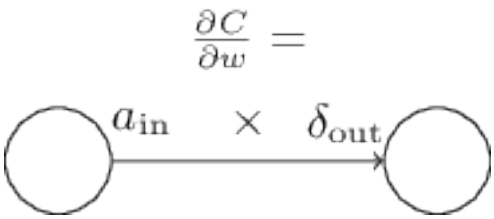
network: In particular:

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

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

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

where it's understood that a_{in} is the activation of the neuron input to the weight w, and δ_{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_{in} is small, $a_{\text{in}} \approx 0$, 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

[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.

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 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

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

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

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

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

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

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

  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

$$\delta^L_j = \frac{\partial C}{\partial z^L_j}. \tag{2.13}$$

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

$$\delta^L_j = \sum_k \frac{\partial C}{\partial a^L_k} \frac{\partial a^L_k}{\partial z^L_j}, \tag{2.14}$$

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

$$\delta^L_j = \frac{\partial C}{\partial a^L_j} \frac{\partial a^L_j}{\partial z^L_j}. \tag{2.15}$$

Recalling that $a^L_j = \sigma(z^L_j)$ the second term on the right can be written as $\sigma'(z^L_j)$, and the equation becomes

$$\delta^L_j = \frac{\partial C}{\partial a^L_j} \sigma'(z^L_j), \tag{2.16}$$

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 $\delta^l_j = \partial C / \partial z^l_j$ in terms of $\delta^{l+1}_k = \partial C / \partial z^{l+1}_k$. We can do this using the chain rule,

$$\delta^l_j = \frac{\partial C}{\partial z^l_j} = \sum_k \frac{\partial C}{\partial z^{l+1}_k} \frac{\partial z^{l+1}_k}{\partial z^l_j} = \sum_k \frac{\partial z^{l+1}_k}{\partial z^l_j} \delta^{l+1}_k, \tag{2.17}$$

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

$$z^{l+1}_k = \sum_j w^{l+1}_{kj} a^l_j + b^{l+1}_k = \sum_j w^{l+1}_{kj} \sigma(z^l_j) + b^{l+1}_k. \tag{2.18}$$

Differentiating, we obtain

$$\frac{\partial z^{l+1}_k}{\partial z^l_j} = w^{l+1}_{kj} \sigma'(z^l_j). \tag{2.19}$$

Substituting back into (2.17) we obtain

$$\delta^l_j = \sum_k w^{l+1}_{kj} \delta^{l+1}_k \sigma'(z^l_j). \tag{2.20}$$

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 $z^l = w^l a l - 1 + b^l$ 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

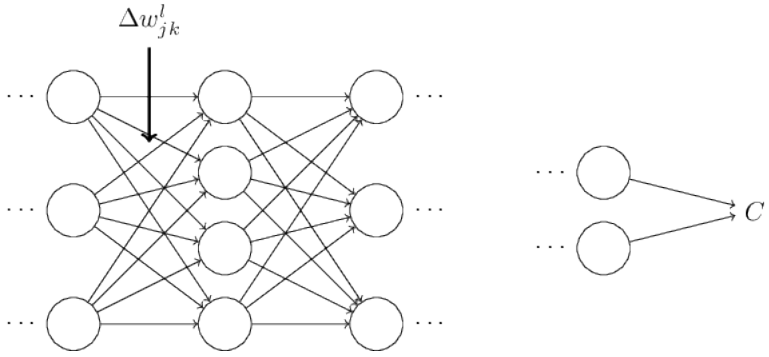
[5]This should be plausible, but it requires some analysis to make a careful statement. It's plausible because the dominant computational cost in the forward pass is multiplying by the weight matrices, while in the backward pass it's multiplying by the transposes of the weight matrices. These operations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

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.

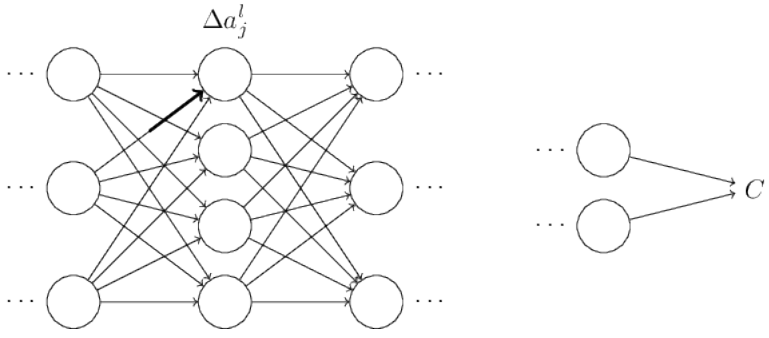
2.9 Backpropagation: the big picture

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 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.

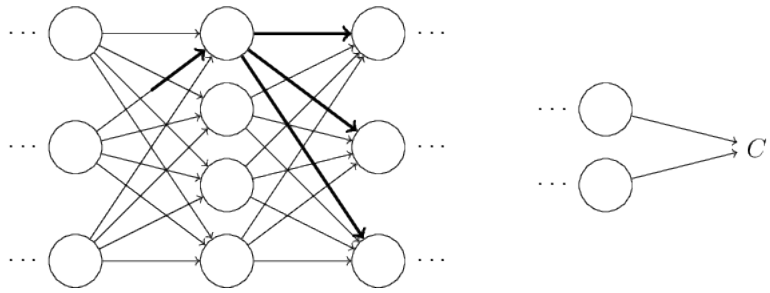
To improve our intuition about what the algorithm is doing, let's imagine that we've made a small change Δw^l_{jk} to some weight in the network, w^l_{jk}:



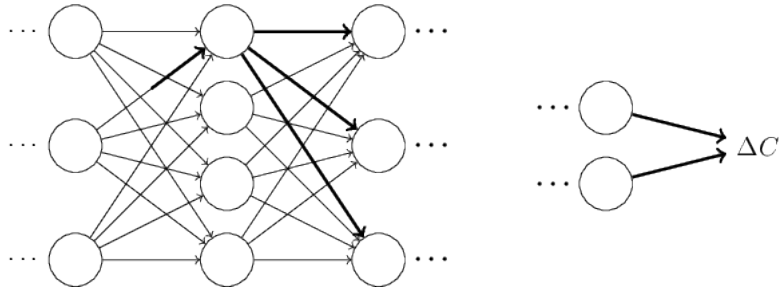
That change in weight will cause a change in the output activation from the corresponding neuron:



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

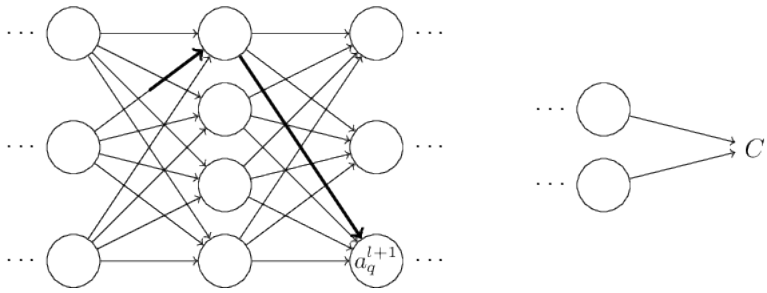
$$\Delta C \approx \frac{\partial C}{\partial w^l_{jk}} \Delta w^l_{jk}. \tag{2.22}$$

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:

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \Delta a_j^l. \tag{2.24}$$

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

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

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.,

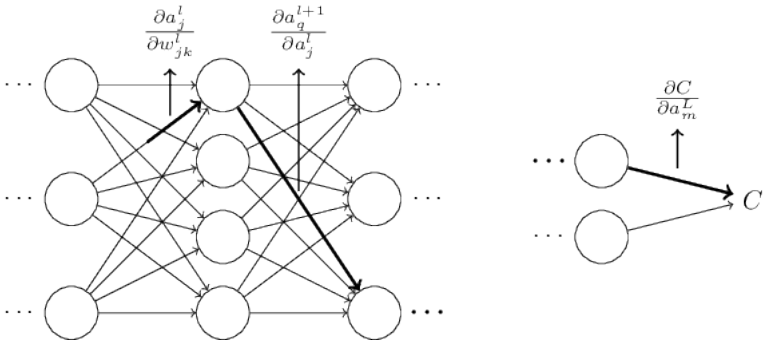
$$\Delta C \approx \sum_{mnp\ldots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l, \tag{2.27}$$

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

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

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:



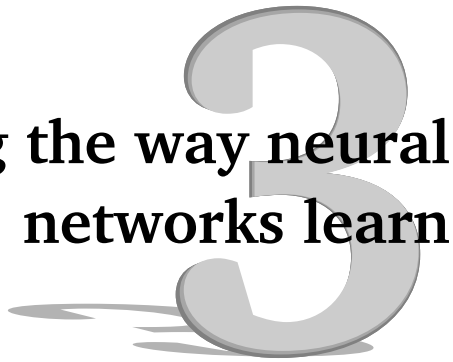
What I've been providing up to now is a heuristic argument, a way of thinking about what's going on when you perturb a weight in a network. Let me sketch out a line of thinking you could use to further develop this argument. First, you could derive explicit expressions for all the individual partial 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

[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.

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.



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.

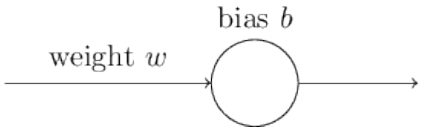
Of course, we're only covering a few of the many, many techniques which have been developed for use in neural nets. The philosophy is that the best entree to the plethora of available techniques is in-depth study of a few of the most important. Mastering those important techniques is not just useful 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

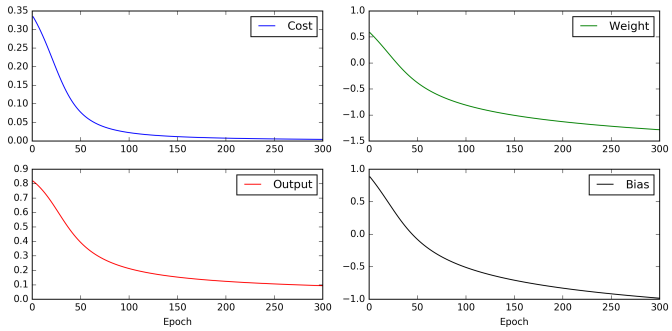
too low. I got confused, and couldn't continue until someone pointed out my error. I was very embarrassed. Yet while unpleasant, we also learn quickly when we're decisively wrong. You can bet that the next time I played before an audience I played in the correct octave! By contrast, we learn more slowly 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:

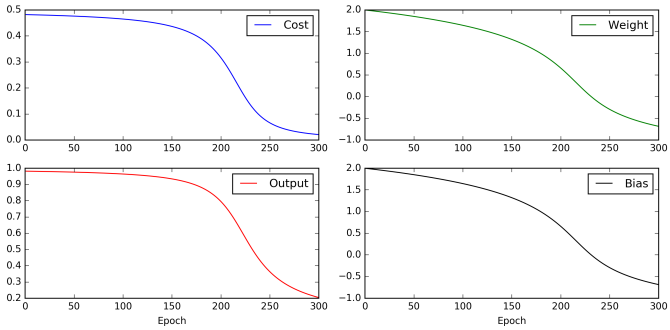


We'll train this neuron to do something ridiculously easy: take the input 1 to the output 0. Of course, this is such a trivial task that we could easily figure out an appropriate weight and bias by hand, without using a learning algorithm. However, it turns out to be illuminating to use gradient descent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

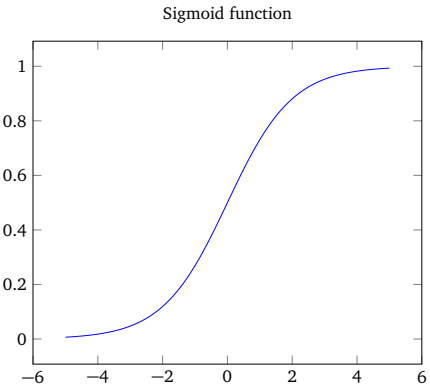
$$C = \frac{(y-a)^2}{2}, \tag{3.1}$$

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

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

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

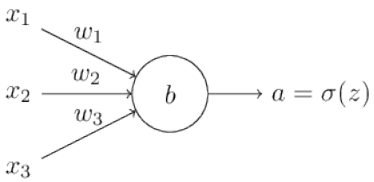
where I have substituted $x = 1$ and $y = 0$. To understand the behaviour of these expressions, let's look more closely at the $\sigma'(z)$ term on the right-hand side. Recall the shape of the σ function:



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.

3.2 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

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

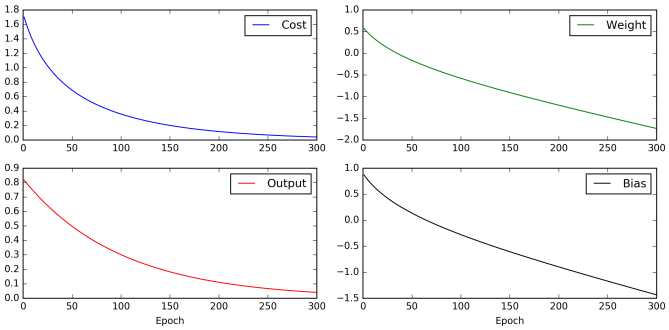
[1]To prove this I will need to assume that the desired outputs y are all either 0 or 1. This is usually the case when solving classification problems, for example, or when computing Boolean functions. To understand what happens when we don't make this assumption, see the exercises at the end of this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

Again, this avoids the learning slowdown caused by the $\sigma'(z)$ term in the analogous equation for the quadratic cost, Equation (3.3).

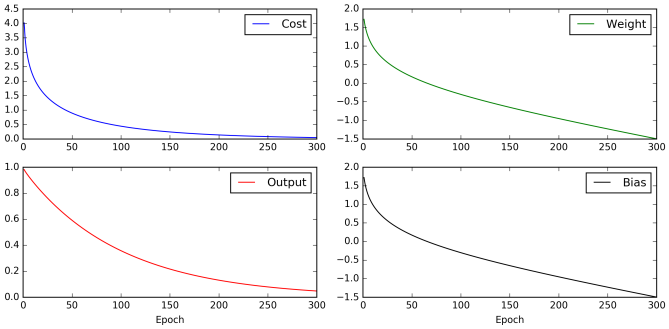
Exercise

- 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

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

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 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

$$\frac{\partial C}{\partial w^L_{jk}} = \frac{1}{n}\sum_x a^{L-1}_k (a^L_j - y_j) \tag{3.15}$$

$$\frac{\partial C}{\partial b^L_j} = \frac{1}{n}\sum_x (a^L_j - y_j). \tag{3.16}$$

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.

3.3 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

[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.

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 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.